Department of Computer Science

An Investigation into the Performance of Modern Parallel Programming Models using TeaLeaf

Matthew James Martineau

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering

September 2015 | CSMSC-15

Declaration:

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Matthew James Martineau, September 2015

EXECUTIVE SUMMARY

This research project details the development and optimisation of new implementations of the TeaLeaf mini-app, and performance tests them on a range of modern HPC devices. The findings are used to evaluate several modern parallel programming models: *RAJA* and *Kokkos* are new performance portable models that use C++, *OpenMP 4.0* is a new revision of the mature directive-based model that now supports offloading code to a target device, *OpenCL* is a well supported open specification for developing applications capable of heterogeneous execution, and *OpenMP* and *CUDA* are mature device-specific models that are used to show the best possible performance attainable for TeaLeaf on each device.

Both 2D and 3D applications were developed using each model, and a discussion of the process presents some interesting insights into the development benefits of the technologies. Using the set of ports, a comprehensive suite of tests was run on the Blue Crystal phase 3 supercomputer, generating a wealth of performance data. This data is visualised and discussed to present a comparative analysis of the efficacy of each model, exposing which models actually achieve their claims of performance portability and to what extent.

This investigation finds that the best performance is achieved with device-tuned implementations but several modern performance portable models are available that can run heterogeneously with a 5-20% performance penalty. As expected, the less mature technologies lack functional portability, which leaves them at a disadvantage regardless of the architectural or developmental improvements that they offer. Also, OpenMP implementations of TeaLeaf that use Fortran, C, and C++ are used to show that, for smaller mesh sizes, Fortran and C achieve better performance than C++, even if the code is identical.

The Atomic Weapons Establishment (AWE), are supporting this research because they need to understand whether the models labelled as performance portable can be used to update their current scientific codes to take advantage of supercomputing resources. In particular, RAJA and Kokkos are of interest because the US national labs, who are handling similar scientific problems, have developed them specifically for this purpose. Importantly, little evidence exists about their efficacy and this paper presents the first ever side-by-side comparison.

As part of this project a number of valuable contributions have been made:

- Brand new production-quality ports of TeaLeaf in 2D and 3D: **RAJA**, **Kokkos**, **CUDA**, **OpenMP** C & C++, **OpenMP** 4.0. Accessible from https://github.com/UoB-HPC/.
- A discussion of how each model can be implemented (chapter 7), and optimised for portable performance on the tested devices (chapter 9).
- A rare description of how to introduce OpenMP 4.0 into a non-trivial application and achieve reasonable performance on an Intel Xeon Phi Knights Corner coprocessor.
- Raw data from over 15000 test runs is visualised and analysed to demonstrate the performance of each model over a range of problem sizes and devices (chapter 10).
- Improvements to the RAJA alpha implementation, to feed back to the developers.
- An accepted conference poster that will be presented at the International Conference for High Performance Computing, Networking, Storage and Analysis 2015 (SC15).

ACKNOWLEDGEMENTS

A massive thank you to Simon McIntosh-Smith for his expert guidance and support throughout and the HPC group for their valuable weekly input, which has helped make this project a success. I also want to thank Wayne Gaudin and the Atomic Weapons Establishment for enabling and supervising the project, and I hope that the outputs can be of significant benefit to them.

Finally, I would like to thank my wife and family for their understanding, patience and support.

CONTENTS

1 INTRODUCTION 1
1.1 High Performance Computing (HPC) . . . 1
1.2 The TeaLeaf Mini-App . . . 1
1.2.1 The Application . . . 1
1.2.2 The Algorithm . . . 2
1.3 Project Aims and Objectives . . . 3
1.4 Conclusions . . . 3

i BACKGROUND 4
2 MODERN HPC ARCHITECTURE 5
2.1 Central Processing Units (CPUs) . . . 5
2.2 General Purpose Graphics Processing Units (GPGPUs) . . . 6
2.3 Intel Xeon Phi Coprocessor - Knights Corner . . . 7
2.4 Conclusion . . . 8
3 PROGRAMMING MODELS 9
3.1 Kokkos . . . 9
3.1.1 Background and Motivation . . . 9
3.1.2 Programming Model . . . 9
3.1.3 Code Migration . . . 11
3.1.4 Conclusions . . . 11
3.2 OpenMP 4.0 . . . 11
3.2.1 Background and Motivation . . . 11
3.2.2 Programming Model . . . 12
3.2.3 Code Migration . . . 12
3.2.4 Conclusions . . . 13
3.3 The RAJA Portability Layer . . . 13
3.3.1 Background and Motivation . . . 13
3.3.2 Programming Model . . . 13
3.3.3 Code Migration . . . 14
3.3.4 Conclusions . . . 14
3.4 Compute Unified Device Architecture (CUDA) . . . 15
3.4.1 Background and Motivation . . . 15
3.4.2 Programming Model . . . 15
3.4.3 Conclusions . . . 16
3.5 Open Computing Language (OpenCL) . . . 16
3.5.1 Background and Motivation . . . 16
3.5.2 Programming Model Stack . . . 16
3.5.3 Conclusions . . . 17
4 OPTIMISATION 18
4.1 CPUs and Accelerators . . . 18
4.2 GPUs . . . 18
4.3 Parameterisation . . . 19
4.3.1 Per-Device Configuration . . . 19
4.4 Compilers . . . 20

4.5 Non-Portable . . . 20
4.6 Conclusion . . . 21
5 PERFORMANCE EXPERIMENTATION 22
5.1 Measuring Performance . . . 22
5.2 Experimental Conditions . . . 22
5.3 Benchmarking . . . 23
5.3.1 Application Benchmarking . . . 23
5.3.2 Platform Benchmarking . . . 23
5.3.3 Performance Portability . . . 24
5.3.4 Conclusions . . . 24

II DESIGN AND IMPLEMENTATION 25
6 IMPLEMENTATION METHODOLOGY 26
6.1 Version Coverage . . . 26
6.2 Constraints . . . 27
6.2.1 Limiting Variation . . . 27
6.2.2 Portability Requirements . . . 27
6.2.3 Minimum Required Functionality . . . 27
6.2.4 Result Validation . . . 28
6.3 Tools . . . 28
6.4 Conclusions . . . 28
7 DESIGN, PROGRAMMING AND CONFIGURATION 29
7.1 Comprehending TeaLeaf . . . 29
7.2 OpenMP C++ and OpenMP MIC Native . . . 30
7.2.1 Design and Implementation . . . 30
7.2.2 Configuration and Build . . . 30
7.2.3 Evaluation . . . 31
7.3 Kokkos . . . 31
7.3.1 CUDA and Accelerators . . . 32
7.3.2 Configuration and Build . . . 32
7.3.3 Evaluation . . . 33
7.4 RAJA . . . 33
7.4.1 Configuration and Build . . . 34
7.4.2 Evaluation . . . 34
7.5 CUDA . . . 35
7.5.1 Reductions . . . 35
7.5.2 Configuration and Build . . . 36
7.5.3 Evaluation . . . 36
7.6 OpenMP 4.0 . . . 37
7.6.1 Configuration and Build . . . 38
7.6.2 Evaluation . . . 38
7.7 OpenCL . . . 38
7.7.1 Evaluation . . . 38
7.8 Conclusions . . . 39
8 EXPERIMENTAL DESIGN AND PROCESS 40
8.1 Platforms and Devices . . . 40
8.1.1 Conclusions . . . 41
8.2 Test Automation . . . 41

8.2.1 Queuing . . . 41
8.2.2 Performance Logging . . . 41
8.2.3 Performance Visualisation . . . 42
8.2.4 Conclusions . . . 42
8.3 Profiling . . . 42
8.3.1 Bandwidth . . . 43
8.3.2 Reviewing and Reports . . . 44
8.3.3 Conclusions . . . 44
8.4 Test Design . . . 44
8.4.1 Configuration . . . 44
8.4.2 Test Details . . . 44
8.4.3 Conclusions . . . 45

III RESULTS, ANALYSIS AND CONCLUSIONS 46

9 OPTIMISATIONS 47
9.1 TeaLeaf Parameter Tuning . . . 47
9.1.1 Chebyshev Pre Steps . . . 47
9.1.2 PPCG Inner Steps . . . 48
9.1.3 Conclusions . . . 49
9.2 Data Access . . . 49
9.2.1 Conclusions . . . 51
9.3 Model-Specific . . . 51
9.3.1 RAJA . . . 51
9.3.2 OpenMP 4.0 . . . 52
9.3.3 OpenMP . . . 53
9.3.4 Conclusions . . . 54
9.4 Un-Optimisable Discoveries . . . 54
9.4.1 Fortran PPCG . . . 54
9.4.2 OpenCL CPU Outliers . . . 55
9.4.3 Conclusions . . . 56

10 RESULTS 57
10.1 Core Scaling Tests . . . 57
10.1.1 Conclusions . . . 59
10.2 Even Step . . . 59
10.2.1 Conclusions . . . 61
10.3 Power of 2 . . . 61
10.3.1 Conclusions . . . 62
10.4 Comparison by Compiler . . . 62
10.4.1 Conclusions . . . 63
10.5 Fortran vs. C vs. C++ . . . 64
10.5.1 Conclusions . . . 64
10.6 Runtimes by Devices . . . 65
10.6.1 CPU Ports . . . 65
10.6.2 GPU Ports . . . 66
10.6.3 Intel Xeon Phi KNC Results . . . 67
10.6.4 Conclusions . . . 68
10.7 Bandwidth . . . 68
10.7.1 Conclusions . . . 70

10.8 Discussion 70
11 CRITICAL EVALUATION 72
11.1 Design, Development and Configuration 72
11.2 Testing and Visualisation 73
11.3 Optimisation and Results 74
11.4 Conclusion 74
12 CONCLUSIONS 75
12.1 Further Work 75
12.2 Summary 76

IV APPENDIX 77
A ADDITIONAL RESULTS 78
B TEALEAF CODE AND SC15 POSTER 82
C APPENDIX C 87

BIBLIOGRAPHY 88

1

INTRODUCTION

1.1 HIGH PERFORMANCE COMPUTING (HPC)

HPC is undergoing significant growth as science and engineering are increasingly reliant upon large-scale simulations to support their cutting edge progress. In order to take advantage of supercomputing platforms, scientific codes need to be carefully re-engineered to exploit concurrency. Even after an application has been parallelised, portability can be a major issue, with many parallel programming models tying you to a particular device or platform. Scientific applications are often long-lived and monolithic, meaning that they cannot be easily rewritten to take advantage of supercomputing resources, greatly inhibiting their potential [21].

The International Exascale Software Project has been initiated by a number of important national agencies, and supported by the high-end scientific computing community [14]. The project exists to assist the entire HPC ecosystem in achieving exascale computation, or one million trillion floating point operations per second (FLOPS). They stated that computational power needs to improve at the fastest rate possible to support potentially transformative scientific progress, but that current HPC infrastructure is incapable of reaching such targets.

Power has become a serious limiting factor in designing new HPC technologies, leading to a shift towards many-core devices and heterogeneous computing [26]. Heterogeneity, employing diverse processors within a platform, is a priority research direction of the project and an important trend. Many of the world's fastest supercomputers include a mix of CPUs, GPUs and accelerators and as a consequence, applications are becoming harder to develop and maintain.

These factors have created a demand for programming models that will allow scientific applications to take advantage of heterogeneous HPC resources, without having to maintain versions for each device. Whilst there are libraries and standards that enable some level of functional portability, they do not necessarily guarantee performance portability [23].

To balance the portability, performance and programmability of a model is a highly complex task, and as the whole landscape can change in an instant, the changes are notoriously difficult to plan for. Given the prohibitive cost of rewriting scientific applications and the current rapid rate of change, it is imperative that application developers are well informed when they choose a modern parallel programming model, in order to safeguard their HPC investments.

1.2 THE TEALEAF MINI-APP

1.2.1 *The Application*

TeaLeaf is an open source project that belongs to both the UK Mini App Consortium (UKMAC) [52] and Mantevo project [21], a collection of "mini-apps" that are miniature

proxies representing important physics algorithms. The UKMAC represents a consolidated national effort to understand modern technologies and algorithms, that is fed into by Warwick University, Oxford University and the University of Bristol, supported and funded by the Atomic Weapons Establishment (AWE). The Mantevo project, run by Sandia National Laboratories, is an award winning collection of open source applications geared towards analysing high performance computing applications.

Mini-apps support the investigation of optimisation, scalability and performance portability, free from the limitations imposed by attempting such analyses with fully functional scientific applications. Importantly, this supports research into techniques for optimising such codes that can eventually be transferred into real scientific applications.

TeaLeaf, in particular, has been developed by a number of institutions in partnership with AWE. The mini-app strikes a perfect balance, matching the computational complexity of the real scientific codes, whilst being much smaller, which makes it more amenable to experimentation.

Table 1 details those versions of the TeaLeaf application that are relevant to this research project, noting those newly created or redeveloped to support testing in this project.

TECHNOLOGY	TWO-DIMENSIONAL	THREE-DIMENSIONAL
OpenMP	Redeveloped	Redeveloped
OpenCL	Original	Original
CUDA	Redeveloped	New
OpenMP 4.0	New	New
Kokkos	New	New
RAJA	New	New

Table 1: TeaLeaf implementations, where "Original" and "Redeveloped" versions already existed prior to the research project, but the latter has been remade or adapted.

OpenCL, Kokkos, RAJA and OpenMP 4.0 were designed to assist developers in implementing performance portable parallel codes. The CUDA implementation provides a GPU-specific benchmark to compare the others against. Each will be optimised using techniques derived from the literature, avoiding non-portable directives for the performance portable models.

1.2.2 *The Algorithm*

Whilst an intimate understanding of the domain is not essential to the research project, successfully optimising new versions of the application will require an awareness of the scientific problem. The three-dimensional heat conduction equation is a simple partial differential equation:

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right)$$

TeaLeaf solves the heat diffusion equation on a spatially decomposed grid and is characterised by two of the seven Dwarfs [2], structured grid and sparse linear algebra. The simulation stores temperatures in each cell of a mesh and uses a seven point stencil, to

calculate a conduction coefficient from the cell density and averaging across the stencil. To ensure numerical stability the problem is solved implicitly, which makes performance optimisation more challenging than with explicit stencil applications [52].

The scientific code is mature and well tested, and to ensure minimal bias when comparing the implementations, the core logic of the solvers will be conserved.

1.3 PROJECT AIMS AND OBJECTIVES

This project aims to present a groundbreaking comparative performance evaluation of a set of complementary parallel programming models, in terms of their development costs and performance potential on modern heterogeneous devices. In order to achieve this it will be necessary to fulfil the following objectives:

- **Implement new versions of TeaLeaf** - Each parallel programming model will be used to create two new "ports", one to solve 2D problems and one for 3D problems. This will result in a broad collection of new research tools that will be released as open source implementations for future research.
- **Optimise each application** - Through iterative optimisation and testing, each application will be optimised for the best possible performance on supported devices.
- **Collect extensive performance data** - A suite of tests will be designed and implemented in order to collect as much performance data as possible, using modern HPC devices.
- **Quantitatively and qualitatively evaluate the programming models** - The efficacy of each application will be measured by their performance on a range of devices as well as the improvements they offer for ease of development.

Successful completion of the outlined aims and objectives will guarantee insights into the performance portable programming models that can be of real benefit to not just the industrial partner, but any application developer who needs to inform their future HPC investment.

1.4 CONCLUSIONS

The Atomic Weapons Establishment (AWE), the industrial partner for this project, use scientific codes that have been developed over many years and cannot currently take advantage of supercomputing facilities. Their continuing scientific progress will rely upon those applications being re-written or adapted to use a portable parallel programming model, but as the process will be so expensive, it is imperative that their choices are supported with as much evidence as possible.

New models like RAJA and Kokkos are being developed in response to this need but it is essential that research, as is presented in this paper, exposes the costs and penalties associated with any particular choice.

Part I

BACKGROUND

2

MODERN HPC ARCHITECTURE

The programming models evaluated in this research have been developed specifically to handle recent changes in HPC architecture, and so understanding some differences between the architectures will be essential for successfully porting the application.

Modern HPC architecture is becoming increasingly more powerful and parallel at the node-level, and the testing performed during this research project focuses exclusively upon single-node performance. Modern CPUs, GPUs and accelerators are increasing internal parallelism to improve concurrency in a power efficient way, which is making it even harder to write code that can run optimally on all devices.

Rupp et al. [43] concluded that good performance on a device suggested good performance with devices from the same vendor, but not necessarily between vendors. Du et al. [39] takes this point even further and stated that kernels optimised for one vendor will perform poorly on another vendor's architecture.

They further observed that performance was improved for many configurations on newer devices, emphasising limitations with outdated hardware, software and drivers. Importantly, the performance testing for TeaLeaf will utilise fairly up to date hardware, that is present in many of the top performing supercomputers in the world. Also, the testing will only permit limited configurational differences on a per-device basis, such as OpenCL work groups.

2.1 CENTRAL PROCESSING UNITS (CPUS)

Modern CPUs are taking advantage of parallelism by increasing core counts and the size of vector units, as shown in the 8 core Sandybridge CPU in Figure 1.

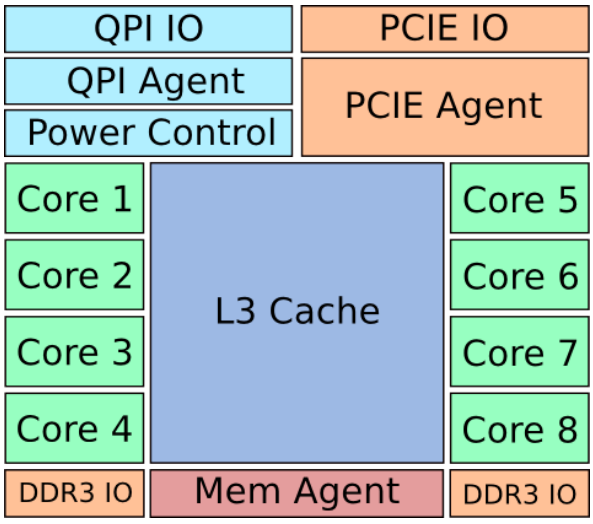


Figure 1: A block diagram of a Sandybridge CPU, redrawn from [24].

Lee et al. [29] suggests that CPUs are designed to be highly reponsive when working on a single process, and need to enable a varied range of tasks. They further explain that the

introduction of special features such as branch prediction, out-of-order execution etc. have greatly improved the performance of the CPU but limit the amount of cores that can reside on a single processor.

2.2 GENERAL PURPOSE GRAPHICS PROCESSING UNITS (GPGPUS)

While historically GPUs were used exclusively as rendering devices, modern GPUs are now being used to handle the compute that would be typically limited to the CPU [29]. As many scientific applications deal with large datasets, there is a trend for applications to not be limited by the arithmetic potential of a device, but by the bandwidth of access to the data.

The CUDA programming guide provided by NVIDIA [9] explained that GPUs were designed for a different purpose to CPUs, specialising for highly parallel and intensive computation. A greater number of transistors are devoted to data processing rather than control flow, and the architecture is optimal for data parallel routines that run the same logic for each element, the single instruction multiple data (SIMD) paradigm [28]. GPUs are designed with high memory bandwidth and many simple processors that work on wide vector units, ideally suited to data intensive applications.

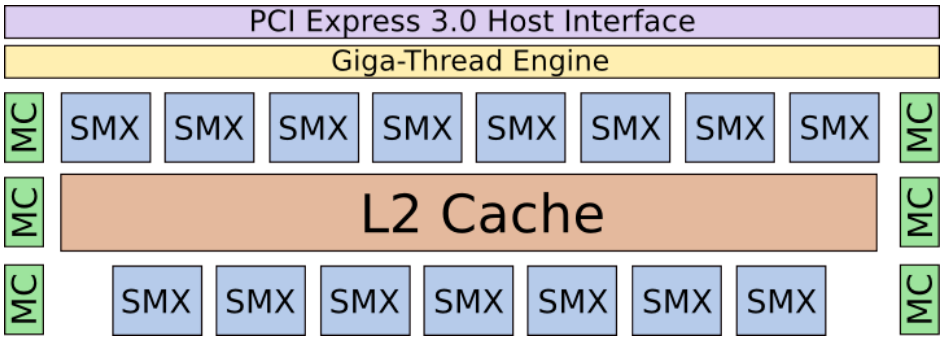


Figure 2: A block diagram of K20 GPU, redrawn from [37]. MC is memory controller and SMX is streaming multiprocessor.

As shown in Figure 2, the K20 GPU contains many streaming multiprocessors, which looks on the surface similar to the Sandybridge CPU layout Figure 1. Of course, Figure 3 shows that each streaming multiprocessor contains a complex hierarchy of cores and memory, which results in the device having 2496 cores in total. Using streaming multiprocessors in this way gives much better performance for data intensive tasks but makes programming for the devices more challenging than the CPU because of the additional memory management.

Kessler et al. [25] commented on the energy efficiency of GPUs, whilst Che et al. [6] reaffirmed the trend of GPUs being used as many-core devices that offer massive parallelism and high memory bandwidth. TeaLeaf is particularly suited to GPU execution given that the problem is a highly parallelisable structured grid problem known to be memory bandwidth limited.

Du et al. [39] suggested that GPUs have gained greater support within the scientific community since support for Fused Multiply-Add (FMA) was introduced, which is more accurate than Multiply-Add and only requires a single rounding step. Further, GPUs have seen improving cache hierarchies and the introduction of Error Correction Codes (ECC), which guards code from random bit-flipping.

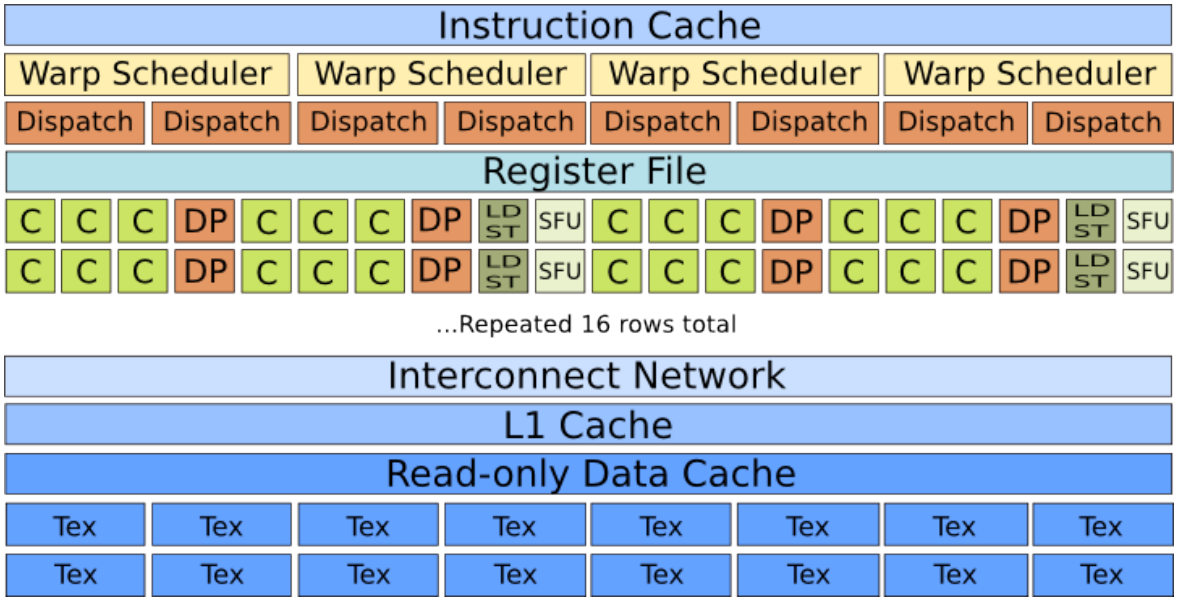


Figure 3: A block diagram of the streaming multiprocessor (SMX) of a K20 GPU, redrawn from [37]. C is core, DP is double precision unit.

2.3 INTEL XEON PHI COPROCESSOR - KNIGHTS CORNER

The Knights Corner (KNC) is the first Intel Xeon Phi, a pioneering coprocessor that represents their entry into the many-core market. As with many first attempts, there have been issues with performance and this has received regular attention in the literature, and so it will be investigated in this research project to see if the results are consistent.

A number of key papers demonstrate that the achievable peak memory bandwidth of the Xeon Phi is far lower than its competitor devices. Teodoro et al. [49] discovered that the effectiveness of the Phi was highly dependent upon the properties of the problem, being particularly suited to applications with regular operations and computational patterns.

While investigating the performance portability of a range of devices, Rupp et al. [43] stated that they were "unable to obtain satisfactory performance for memory bandwidth limited operations using OpenCL on the Intel Xeon Phi". The report noted that Intel engineers have corroborated such claims, but offered no additional insight into the potential causes of the issue.

Demeshko et al. [12] found the Intel Xeon Phi KNC to be the worst performing device when compared to an NVIDIA GPU and Intel Xeon CPU, only achieving a very small speedup compared to the serial code. The problems exhibited with Xeon Phi performance are synonymous with those results seen in the main Kokkos technical paper [15], where the scaling is far less performant for the Xeon Phi. This particular paper mentioned that the Xeon Phis used for testing were pre-production versions, which may have had an effect on the results.

McIntosh-Smith et al. [34] found that for the lattice Boltzmann application, the Xeon Phi was only capable of sustaining 10% of peak memory bandwidth, which they open as an area that needs further research. They stated (1) the OpenCL runtime for Xeon Phi is new (2) the Xeon Phi only achieves 52% idealistic benchmark memory bandwidth.

Interestingly, Saule et al. [45] state that while the theoretical bandwidth is 352 GB/s, which is the figure advertised, the cores and memory controllers are connected by a ring

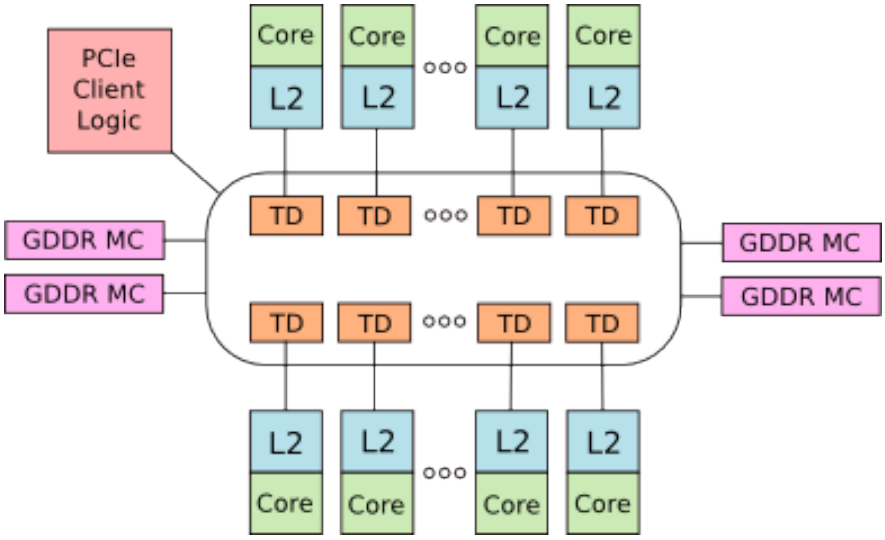


Figure 4: A block diagram of the KNC architecture, redrawn from [7].

network (Figure 4) that can only theoretically achieve 220 GB/s. This may somewhat explain why the performance achieved on the devices are lower than would be expected.

KNC devices can be programmed in two ways, (1) use an offloading framework, which supports offloading atomic units of computation to the device, or (2) compiling an entire application to run on the Intel Xeon Phi, which is called native compilation. This project will compare both options to understand the different performance attainable with each method but it can be noted that in the future, the KNC architecture will be replaced with the drastically altered Knights Landing (KNL) model [20]. The KNL promises to offer a very high bandwidth device that can be programmed in the same manner as a CPU.

2.4 CONCLUSION

Evaluating results from particular devices and architectures can be quite difficult between studies given the multitude of possible configurations for aspects such as drivers, compilers, and interconnects. Through carefully considered data collection and visualisation of those results, this research is able to use the new ports of TeaLeaf to discuss some of these architectural variations. In particular, it will be interesting to show whether the issues with the Intel Xeon Phi KNC are hardware related performance issues or performance portability problems with OpenCL.

3 PROGRAMMING MODELS

Extensive research was required to be able to design each port of the TeaLeaf application and understand the model specific optimisation potential. In general, the programming models have been designed in isolation from each other, meaning that each draws upon a unique set of patterns and practices, a succinct description of which is presented.

3.1 KOKKOS

The Kokkos package is part of the Trilinos project, developed by Sandia National laboratories to provide a modern library-based approach to performance portability. The project emphasises the development of "robust algorithms for scientific and engineering applications on parallel computers".

3.1.1 *Background and Motivation*

Edwards et al. [15], acknowledge two principal techniques that embody the power of the model:

A. Utilising abstraction to perform computation on many-core devices

B. Leveraging the power of C++ templates to provide portable high performance data layout tuning functionality

A Kokkos tutorial by Edwards et al. [16] further emphasised that usability of the library is considered but that the design prioritises portability and performance.

- **Structure of Arrays (SoA) vs. Array of Structures (AoS):** To enable optimal memory usage, data structures may need to be blocked using the AoS pattern for CPUs and coalesced using the SoA pattern for GPUs/accelerators [29, 18].
- **Continual change in HPC:** The landscape of HPC architecture is constantly changing as new strategies are developed to overcome the limitations in existing hardware.

By decoupling the code from the underlying platforms, the library can add support for new devices with minimal disruption to the core codebase. This fosters a write (or rewrite) once approach that promises that even the largest of applications can continue benefiting from the portability as long as back-end implementations are maintained.

3.1.2 *Programming Model*

The programming model provides a range of generic abstractions that allow the user to create new codes or port existing applications:

- **Execution and Memory Spaces**: The library makes a distinction between execution space and memory space, to support GPUs and accelerators which need to have distinct memory from the host CPU. The library handles much of the interaction between these spaces but some explicit initialisation and data transfer semantics are handled by the developer.

- **Data Structures:** This abstraction supports mixing dynamic and compile-time dimensions for optimisation, as well as copy semantics analogous with the C++ std::shared_ptr to avoid complex ownership constraints.

```
// Array in memory space Device with dimensions (N,M,256,256)
View< double**[256][256], Device > foo("foo",N,M);
```

- **Functors:** The library utilises C++ class constructs called functors, where the function operator is overloaded and encapsulates the core functional logic. This pattern also allows data objects local to the class and some number of supporting callback functions. The tutorial [16] suggested that the functor pattern was chosen to enforce C++1998 standard compliance. It was argued by Hornung et al. [22] that compiler optimisation of functors with function templates is currently "deficient", presenting a risk to portability.

```
// Simple functor that scales elements of array by a factor
template< class Type, class Device >
class ScaleFunctor {
    public:
        typedef Device deviceType;

        // The core computation that scales each element
        KOKKOS_INLINE_FUNCTION;
        void operator()(int index, double factor) const {
            y(index) = y(index) * factor;
        }

    // The array held in Device memory
    View< Type*, Device > const y;
}
```

- **Parallel Execution:** Two key data parallel execution operations are provided: **parallel_for** which encapsulates iterative execution and **parallel_reduce** which allows aggregation of data using some function (e.g. sum).

```
// Abstract implementation of for loop that executes functor
parallel_for(n, AddFunctor< double, Cuda >());
```

Underpinning the semantics is an implementation that uses C++ template meta-programming to rewrite the functors into device-specific code. An interesting point of consideration is that this model, as with all performance portable models, relies upon vendor-specific implementations. In contrast to OpenCL however, the implementations will be maintained by a third-party, leaving long term support at risk until a strategy is outlined for continual support improvement.

3.1.3 *Code Migration*

The following code migration strategy [15] has been adopted while porting TeaLeaf to utilise Kokkos:

1. Manipulate the data structures to use the Kokkos View template.
2. Introduce functors for each computational unit, either relocating logic or creating functor wrappers.
3. Add explicit memory management of the appropriate memory spaces in order to enable GPU execution.
4. Perform application specific optimisation for scalability.

3.1.4 *Conclusions*

The Kokkos library aims to tackle the performance portability problem by abstracting the complexity away from the developer, shielding them from non-portable code artefacts. The library uses template meta-programming to allow applications to be compiled on different devices, with optimisations automatically applied. By using the code migration strategy, and investing significant time, it should be possible to take a code and port it once, making future development easier, and enabling heterogeneous execution.

Although the syntax will be familiar for an experienced C++ developer, the framework requires knowledge that could be considered specialist to others and forces developers to utilise a single language, which may be too constrictive.

3.2 OPENMP 4.0

OpenMP is a directive-based programming model that is widely adopted for parallel programming targeted at CPUs in shared memory environments. The new standard, OpenMP 4.0 [4], introduces a number of directives that are designed to allow portability beyond CPUs.

3.2.1 *Background and Motivation*

The new standard, OpenMP 4.0, has brought changes that are indicative of the shifting landscape of HPC, incorporating a new interface to handle heterogeneous hardware [30]. As discussed in subsection 3.4.1, GPUs and accelerators are being increasingly adopted in new HPC platforms because they can offer a number of benefits when compared to CPUs.

The current consensus is that OpenMP 4.0 offers a much higher-level programming model than the other main options, OpenCL and CUDA, and allows an easier transition of a codebase from sequential to parallel [54, 30]. Ozen et al. [38] remarked that an important consequence of OpenMP 4.0's design is a greater reliance on the compiler to produce optimised initialisation code, kernels, data transfers etc.

3.2.2 *Programming Model*

The execution model takes directions from the host to offload computationally expensive operations to an accelerator device [54]. A recent paper by Dietrich et al. [13] suggested that while OpenMP 4.0 includes computational offloading, it does not guarantee that it takes advantage of the targeted device's capabilities. It can also be noted that there is currently only limited compiler support for the offloading, which means that the TeaLeaf OpenMP 4.0 version can only be tested on Intel Xeon Phi KNCs.

A brief description of some of the main syntax is presented, where Figure 5 demonstrates a basic loop with reduction:

```
! Declare the 'solve_kernel' as a target function
!$omp declare target(solve_kernel)

! Synchronise the global variable 'complete' with the device
!$omp target update to(complete)

! Offload a region of code to the target device
!$omp target map(tofrom: buffer)
    ...   ! Loops and computational functions
!$omp end target map
```

Figure 5: Example of Fortran OpenMP 4.0 syntax [4].

- **OMP DECLARE TARGET(function):** Notifies the compiler that a function, which may have already declared itself as a target, will be offloaded from the current scope.
- **OMP UPDATE TO(global variable):** A variable that has been declared can be updated using this clause, to ensure consistency between target invocations.
- **OMP TARGET MAP(direction: array):** The target region is a section of code, which can contain some number of loops and function calls, all of which are offloaded to the device. Multiple mapping clauses with different directions can be declared to enable data transfer. As discussed by Wienke et al. [54], the memory model does not assure coherence between operations executed in different threads.

While there are a number of other directives available to handle offloading to Single Instruction Multiple Data (SIMD) targets [38], offloading is currently only supported by the Intel Xeon Phi KNC.

3.2.3 *Code Migration*

Code migration has not been described in any depth in the available literature but several examples [30, 38] demonstrate that OpenMP 4.0 is loop centric and so migration will require assessing each of the core application loops individually. Throughout the research project, optimisation diverts the final port away from a loop-centric architecture and the results of this optimisation process is discussed in detail in subsection 9.3.2.

3.2.4 *Conclusions*

OpenMP 4.0 is an intuitive programming model that appears to have a relatively low impact on the core logic of a codebase. Wienke et al. [54] noted that there is currently a lack of adoption, representing a major risk in terms of performance portability that can only be improved with maturity. Once vendor support improves, it will more than likely represent an attractive choice to application developers as OpenMP is an already established and popular technology.

3.3 THE RAJA PORTABILITY LAYER

The RAJA programming model is a brand new abstraction layer designed by Lawrence Livermore National Laboratories (LLNL) to improve performance portability of advanced simulation and computing (ASC) codes.

3.3.1 *Background and Motivation*

The key technical paper by Hornung et al. [23] is the primary source of information regarding the RAJA portability layer and outlined two core goals:

A. Abstract "non-portable compiler and platform-specific directives", and other implementation details, insulating application developers.

B. Make it easier for application developers to tune data layout and memory access for optimal operation on diverse memory hierarchies.

Their perspective on HPC in general is that within-node or "fine-grained parallelism" has been relatively under utilised by scientific application developers. This is becoming a significant inhibitor for performance portability as HPC technologies are adopting greater levels of node-level parallelism including widening vector units and increasing core counts [23, 34, 42, 31].

They suggest that organising and controlling memory locality is an essential step in porting serial scientific applications to run on parallel architectures [46, 56]. Decomposing the problem domains into smaller units allows threads to have improved utilisation of shared data caches, but can lead to non shared data and "domain management operations" saturating the available resources. RAJA makes it easier to perform "chunking", allowing optimisation of instruction and data cache utilisation.

3.3.2 *Programming Model*

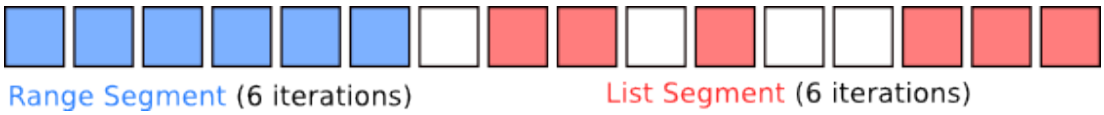
The paper presented the three concepts that are foundational to the design of the model:

- **Separate loop body from traversal:** Separating the loop logic and traversal makes it possible to change iteration sequence, access patterns and device optimisations without affecting the logic. To improve portability and avoid compiler specific issues such as alignment, data type encapsulation is provided.

```
// Encapsulated data types
Real_ptr x;
Real_type a;

// A simple scale by factor loop
for_all< exec_policy >(IndexSet, [&] (Index_type i) {
    x[i] = x[i] * a;
});
```

- **Partition iteration space into work units (Segments):** Abstracting access patterns into "Segments", which fetch data using different access "strategies". When different access patterns are required for a single operation, dividing memory access patterns into similar types allows the strategies to be handled separately, potentially in parallel. Optimal memory access is highly variable between architectures and an agreed inhibitor to portable performance [48].



- **Segment dispatch and execution (Indexsets):** Indexsets are the final abstraction that tie together the previous features by allowing segments to be combined based on type and dispatched for execution using a loop template. Indexsets represent a policy, e.g. "Dispatch segments in parallel and launch each segment on either a CPU or GPU as appropriate", and can be directly created by developers.

The paper noted that the CUDA compiler *nvcc* does not currently have support for lambda functions, which means that there is currently no option for running RAJA on GPUs.

3.3.3 *Code Migration*

Hornung et al. [23] described their experiences porting several applications to use RAJA. It has been possible to devise a general code migration strategy from their discussion:

1. Consider the potential dispatch and execution policies for the application.
2. Determine the Segments appropriate to the domain.
3. Combine the Segments into Indexsets and write initialisation code for these constructs.
4. Replace each loop to be parallelised with an appropriate lambda statement.

3.3.4 *Conclusions*

The portability layer heavily depends upon C++11 compiler support because of the extensive use of lambda functions. Unfortunately, this is regularly outlined as a limitation of the design and poses a significant cause for concern regarding portability [22].

Several features support optimising code without affecting the core logic, and provide very intuitive interfaces that could result in cleaner and more resilient code. RAJA's design includes a number of promising concepts geared towards performance portability, but

the current lack of C++ lambda support [22] means that it cannot provide portability to accelerators or GPUs.

3.4 COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

CUDA is a mature parallel computing platform developed by NVIDIA to allow application developers to offload computation to their GPUs, and should allow the greatest possible performance on NVIDIA devices.

3.4.1 *Background and Motivation*

NVIDIA created CUDA in 2006 [9] and it actually comprises not just a programming model but an entire platform for parallel computation on their enabled GPU devices. As a pioneering technology, CUDA was designed to be easy to learn by application developers, supporting languages such as C. The model does not abstract the GPU architecture but instead uses it to enforce decomposition of problems for task, data and thread parallelism.

Because of the nature of the technology, there are many low-level features and only NVIDIA devices are supported, meaning the model is non-portable by design.

3.4.2 *Programming Model*

Whilst investigating the effectiveness of CUDA for a 2D stencil Jacobi-solver, Cecilia et al. [5] presented some key details of CUDA:

- Local resources are shared amongst threads, with each core of a streaming multiprocessor performing the same operations on individual data elements.
- To instruct the GPU on which instructions to run, the application developer can create kernels (Figure 6). The developer can structure the kernels onto a grid, which decomposes the problem domain into sets of thread blocks that are isolated from each other, with each running on an individual multiprocessor and only sharing global memory.

```
// Simple kernel that scales elements of array by a factor
__global__ void ScaleFunctor(const int a, double* x) {
    // Access on a 1D grid
    const int gid = threadIdx.x+blockIdx.x*blockDim.x;

    // Scale the elements in global memory
    x[gid] = x[gid] * a;
}
```

Figure 6: A simple CUDA kernel

- Lee et al. [28] described that there are a number of memory spaces other than *global* and *local*: (1) *registers* are the lowest latency memory, are accessible only by individual threads and have a finite limit per thread, (2) *texture memory* is similar to global memory except read-only and provides convenient indexing and (3) *constant memory* is optimised for broadcasting data to multiple threads.

The CUDA programming guide [9] offers specific details regarding the differences between different hardware versions, as newer compute devices support improvements to features such as data coalescing. Dawson et al. [10] noted that since the Fermi architecture of NVIDIA GPUs, global memory access is now loaded into 128-byte L1 and 32-byte L2 cache lines. Also, they explained that, for NVIDIA GPUs, threads are scheduled into 16-32 thread sub-blocks called warps. It is possible to organise data structures and configure parameters to take advantage of this important vendor-specific detail.

3.4.3 *Conclusions*

As CUDA is intrinsically non-portable, it will serve purely as a benchmark with which the other implementations can be compared. After rigorous optimisation this implementation can demonstrate a best achievable performance for NVIDIA GPUs, which are also targetted by Kokkos and OpenCL. Without performance portable models it would be necessary to maintain some number of architecture-specific implementations [56, 23]. As this research project develops such implementations from scratch, it will be possible to expose the complexity inherent with developing device-specific models and demonstrate the potential for cost reduction when using performance portable technologies.

3.5 OPEN COMPUTING LANGUAGE (OPENCL)

Although an OpenCL implementation of TeaLeaf already exists, it will be tested in the same manner as the other implementations and ammendments will be made to ensure that the final version conforms to important constraints outlined in section 6.2.

3.5.1 *Background and Motivation*

OpenCL is an open standard for writing applications that can execute on heterogeneous devices, which was released in 2008 [36]. The OpenCL standard [19] states that the approaches to programming CPUs and GPUs are very different, especially the difference in memory models and use of vector operations. OpenCL was developed to offer a low level but portable abstraction that can support both data and task parallel programming models.

Tillet et al. [50] stated that the growing selection of programming models designed for many-core architecture actually limits the ability for architectures to standardise. They expressed this as a motivation for improving adoption of OpenCL, to improve the prospects of standardised performance portability.

3.5.2 *Programming Model Stack*

- **Platform Model** The model represents a host that interacts with some number of devices, with each device containing compute units, which in turn contain processing elements [19].
- **Execution Model** Code that is executed on the host and devices have isolated execution spaces, where kernels represent work that can be completed by a device. Kernels operate within a particular context established by the host code, which connects devices, kernels, program source and variables. Device-specific *command queues* can be

created that allow work to be queued into the device by the host. The queues accept groups of work items called work groups, which are executed on a compute unit, with each work item being handled by an individual processing element [31].

- **Memory Model** The memory model separates the distinct memory regions and objects available to the host and each device that share a context. Similarly to CUDA this includes a distinction between host and device memory, as well as some hierarchy of memory on the devices that relates to the way data is mapped to threads in GPUs. Stone et al. [47] suggest that the actual hardware mapping for these memory types is implementation-specific, for instance local memory could live in low latency local register files for one device and in high latency global memory for another.

```
// Simple kernel to scale an element by a factor
__kernel void ScaleFunctor(
        const int width,
        const int a,
        __global__ double* x)
{
    // Find global id
    const int gid = get_global_id(0)+get_global_id(1)*width;

    x[gid] = x[gid] * a;
}
```

Figure 7: A simple OpenCL kernel.

Because of this hierarchy of abstraction, OpenCL requires significantly more boilerplate code than the other models, most of which don't require any at all. This includes setting up platforms, contexts, command queues, all kernel arguments, and potentially many more abstractions. The kernels actually look very similar to CUDA kernels and an example is provide in Figure 7.

3.5.3 *Conclusions*

The OpenCL standard [19] represents the ideal that vendors should focus upon a single open standard, which could in the long term solve many of the current performance portability problems. As the model has matured, many vendors have created implementations [35] and so good functional portability is already achievable by using OpenCL.

In spite of this, while the standard is very popular, it has not dominated to the extent that might have been expected given its capabilities. This more than likely comes down to two major factors: (1) the standard provides a broad set of generic features, which may make it more complicated than other models, and (2) some research suggests that OpenCL is not necessarily performance portable [18, 31]. Through reviewing the code and performance testing the implementation on the full complement of devices, a discussion is presented relating to these points.

4 OPTIMISATION

Optimisation enables performance, but it can inhibit performance portability, as non-portable optimisation can stop code working on new devices. It is important to understand the available techniques to make each implementation as fast as possible, and determine those optimisations that should be avoided for portability.

4.1 CPUS AND ACCELERATORS

CPU optimisation is well understood, with compilers and technologies like OpenMP having the capacity to create highly optimised code with little developer input. Lee et al. [29] stated that for memory bandwidth bound problems there are still a number of issues that need to be considered on the CPU. CPUs hide latency with cache and so large problems can spill into main memory, inflating the latency. In general, because CPUs have low memory bandwidth compared to GPUs, techniques such as tiling are important to improve cache utilisation. They suggest that as TeaLeaf has structure-of-array (SoA) data structures it should be able to take full advantage of SIMD.

Mallinson et al. [31] found that OpenCL performance for their application was generally unacceptable on CPU architectures prior to optimisation. In particular, they observed that vectorisation played a significant role in this performance issue, and Intel vectorisation reports were utilised to determine the root cause and fix it.

4.2 GPUS

GPUs are more complicated to develop for than CPUs, and have many specialised functions that can be considered when optimising applications. Most importantly, it is emphasised that for memory-bandwidth bound applications the key to GPU optimisation is organising data for lower latency access [39, 28, 8]. To improve performance, GPUs utilise data coalescing, where threads in a group access adjacent chunks of memory (Figure 8). In order to take advantage of coalesced global memory access, the developer must try to structure data into contiguous blocks and carefully consider locality [9].

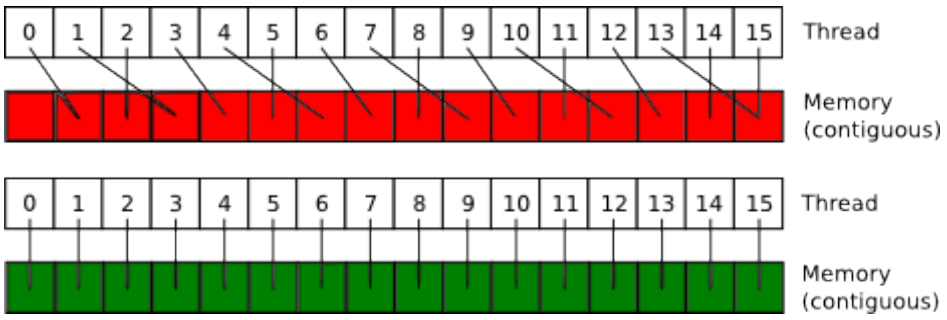


Figure 8: Non-coalesced memory access (top) coalesced memory access (bottom).

Yan et al. [55] described that host-to-device communication occurs across an interconnection bus, generally PCI-E, and the bandwidth is significantly lower than the on-device bandwidth. A highly effective optimisation involves reducing the amount of data that needs to be transferred across this bus. This can generally be achieved by copying all data needed by a device during initialisation and minimising the data transfer between devices through careful buffering.

Godwin et al. [17] described off-chip or global memory as the next highest latency memory with a high bandwidth that is generally difficult to maximise. They suggested that improving bandwidth would not guarantee improved performance if threads pull additional data across the bus but don't use it.

Occupancy is an important aspect of GPU optimisation, where there is a limit to the resources available to a multi-processor [8]. The more shared memory etc. that is utilised, the fewer threads can actually be run at any given time which limits the ability for the GPU scheduling mechanism to saturare the device and achieve maximum memory bandwidth.

It is recommended that, where possible, algorithms are reconfigured so that they exhibit maximal data-parallelism to exploit the strengths of the GPU architecture [10, 5]. This will avoid problems with branch divergence, which is where threads within the same group follow different conditional paths, causing those threads to be executed sequentially and stalling progress. TeaLeaf is already data-parallel, as it uses a structured grid, which are inherently data-parallel, and has been carefully architected to reduce branch divergence.

4.3 PARAMETERISATION

Much modern research is focussing upon making an application optimal on multiple devices by tuning the parameters available for an application and device, potentially automatically during compilation. Auto-tuning is not utilised in this project because the parameters tuned are all independent and therefore provide no search space, but the associated concepts and techniques demonstrate how an application can be tuned for the best possible performance on a device, which will be very important to achieve good performance.

4.3.1 *Per-Device Configuration*

Rupp et al. [43] found that all devices are capable of achieving high performance, given the correct configuration. They stated that performance portability can be achieved between GPU and CPU with a single shared configuration, but around a 10% performance penalty will be likely on some devices. This research outlined a number of optimal settings for the OpenCL work groups parameters, for instance use work group size of 128 or 256 [34].

In other research, Zhang et al. [56] investigated the performance portability of a number of benchmarks using OpenCL. Their methodology tuned available configuration parameters, such as tiling sizes and data layout, to determine the difference between performance portable applications and device-optimised applications. Several times they suggested that the performance portability of OpenCL is "poor" until parameter tuning has been applied, which is a strong statement that does not seem to be substantiated by significant data and does not agree with other research [34, 31].

The TeaLeaf application has a small number of parameters including problem density, and the number of steps run during the Chebyshev and PPCG phases, each of which is a potential target for tuning. There are also parameters specific to the programming

models such as the work group size (for OpenCL), number of threads per block (for CUDA). Determining the parameters available for a particular parallel programming model relies upon an individual analysis of each application.

4.4 COMPILERS

As HPC technologies improve, compilers play an increasingly important role in the optimisation of applications. Compilers can perform automatic optimisations of arithmetic operations and vectorisation of loops, all of which contribute to good performance on particular devices. Kogge et al. [26] discuss the relation between new programming models and their ability to support compilers in performing optimisations. They do also recognise that compilers cannot handle all optimisations and so programming models will often have to expose explicit directives to guide the compilation. In particular, they express that compilers will need guidance in reducing expensive data movement operations.

There are a many options for compiling for x86 architecture, such as the Intel, and GNU compilers. However, this choice is more limited for GPUs and accelerators, for instance NVIDIA GPUs rely on the proprietary nvcc compiler [9]. Each compiler will allow a set of compiler optimisation flags to be provided, which means that there is a large space of potential combinations. Actively considering the effect of different compilers on the performance of the parallel programming models will be important for evaluating the optimal configuration on each architecture. The CPU testing will primarily use the Intel compiler suite, but some investigation will be performed into the differences in performance achieved when compiling with the GNU compiler suite.

4.5 NON-PORTABLE

Some optimisations are intrinsically non-portable and some have different variations for different devices. Understanding the optimisation strategies that inhibit performance portability will be particularly important for evaluating the new versions of TeaLeaf.

Du et al. [39] raised the issue of OpenCL implementation-specific optimisations, that are added by particular vendors. These are highly non-portable making them a poor choice for creating flexible and portable applications. Further, Liao et al. [30] emphasised that managing multiple device-specific kernels significantly increases complexity of the codebase and, if maintained within a single application, compiler and runtime support. For TeaLeaf, a code review will be performed on the existing OpenCL implementation to ensure that such optimisations are not included in the source prior to testing.

Mallinson et al. [31] proposed creating individual reduction operations for both CPU and GPU, allowing fine-grained optimisations of these important operations. This is an interesting point because of the potential performance improvement that can be gained by creating non-portable versions, for minimal conditional code. This optimisation can have a significant effect on the performance, but can often be made portable by using tricks such as pre-processor macros to switch the implementations at compile time.

When optimising the CUDA implementation, there is no restriction that portability must be protected because it is not a portable framework. NVIDIA GPUs enable certain optimisations because of the way that threads are mapped into groups of 32 called warps [10], which may afford some additional device-specific performance.

The performance portable implementations proposed for TeaLeaf all have the option for non-portable optimisations to be introduced at a code level to account for fringe cases: Kokkos can be written without correct data marshalling which may inhibit it from working on GPUs, and RAJA allows template specialisation where the developer can supply highly non-portable artefacts. Within reason, such optimisations will be avoided so that testing can be performed on all devices succesfully.

4.6 CONCLUSION

In order to achieve high performance heterogeneous execution for an application there are several conflicting opinions: some advocating abstraction as a solution [15, 23, 56], and others relying upon autotuning [39, 56, 50], or a best fit configuration that exhibits some performance penalty across devices [43, 34].

Optimising code in a portable manner is a complicated process that often requires splitting code with conditional clauses, potentially so that irrelevant code is ignored by the pre-processor. A highly performance portable framework should be able to automate or abstract such optimisations in order to take full advantage of the device, resulting in optimal code regardless of the device. This project will attempt to optimise each port as far as possible without reducing functional portability so that the final results are representative of the potential performance of each framework.

5 PERFORMANCE EXPERIMENTATION

5.1 MEASURING PERFORMANCE

The data collection phase of this research project will obtain performance metrics for all available platforms and implementations of TeaLeaf. It is imperative that a performance measurement strategy is decided so that the evaluation is consistent and appropriate.

5.2 EXPERIMENTAL CONDITIONS

McIntosh-Smith et al. [34] measured the performance of structure grid codes on a range of different CPUs, GPUs and accelerators. They outlined key methodological constraints:

A. Keep mesh size fixed.

B. Make no changes to code between platforms (host or kernel).

C. Work-group/thread-block size kept fixed or decided by the runtime.

D. Minimise the configuration changes when running on different platforms.

E. Maintain a consistent level of optimisation between implementations.

Constraints A and B are essential for removing bias from the experiments and enforcing consistent measurement. Constraints C and D are complementary, removing all parameterisation from the experimentation, however other studies [56, 39, 43, 50] have shown that parameterisation can make a significant difference to performance portability (section 4.3). In spite of this McIntosh-Smith et al. [34] observed that, for OpenCL structured grid applications, it is possible to develop code that is performance portable whilst adhering to these constraints.

Their argument is that OpenCL has reached a reasonable level of maturity and is now capable of demonstrating true performance portability. The TeaLeaf OpenCL implementation, which has been developed by their group, should exhibit the same performance portability profile and provide a suitable benchmark for the new TeaLeaf implementations.

Their research found that, for one application, OpenCL exhibited 15% improved performance compared to equivalently optimised CUDA implementations. This contradicts the findings of Mallinson et al. [31], who compared OpenCL against device-tuned code and consistently demonstrated a small performance penalty for OpenCL. This may be explained by the differences in hardware architecture utilised for the studies or maturity of drivers used for the testing.

Contrary to constraint E, this project proposes that full optimisation across all models can offer a more fair perspective of the performance of each model. If certain optimisations are excluded it may present a biased representation of one model over another, as each portable parallel programming model may be able to add device-specific optimisations during compilation. This will inflate the observed performance of those performance portable models if the optimisations are avoided when developing the device-specific implementations.

5.3 BENCHMARKING

Consistent and reliable performance results are essential for evaluating the programming models, and the chosen measures must avoid favouring one implementation over another.

5.3.1 *Application Benchmarking*

Whilst evaluating a performance portable lattice Boltzmann code in OpenCL, McIntosh-Smith et al. [33] provided a recommended methodology for testing performance portable code across heterogeneous platforms. They suggested that the performance should not just be measured in terms of absolute runtime but also in terms of performance relative to some applicable peak. There are a number of different metrics that can be considered and it is important to understand which are most relevant to this research project:

- **Speedup:** Measured as a multiplicative factor of change relative to some, typically serial, benchmark of runtime [31, 48, 18].
- **Floating point operations per second (FLOPS):** Entails counting the number of floating point operations that an algorithm performs and then calculating how many are performed per second [33, 39].
- **Memory Bandwidth:** Generally this consists of calculating the amount of data an application reads and writes, converting application wallclock runtime into bandwidth [34, 56, 39]:

Memory bandwidth is the most appropriate measure of performance for TeaLeaf as it is a memory bandwidth bound problem. Rupp et al. [43] suggested that measuring memory bandwidth relative to the vendor specified peak bandwidth may not provide the same insights as comparing it to the achievable peak value. The paper demonstrated a vast disparity between these two figures, with the Xeon Phi achieving only 19.2% of theoretical peak memory bandwidth. This will be incorporated into the data collection phase by benchmarking each platform for achievable memory bandwidth.

As the comparative analysis will include many different ports that are all solving the same problem, simply comparing the absolute runtime will be the preferred method. Bandwidth and speedup calculations will be visualised to expose interesting patterns in the raw runtime data.

5.3.2 *Platform Benchmarking*

In order for the TeaLeaf bandwidth calculations to be useful, it will be important to understand the peak potential of the particular devices being used in the study [34].

Whilst there are many benchmarks that can offer insights into the peak potential of the TeaLeaf algorithm on a particular device, it is impossible to completely remove bias. For instance, it is difficult to prove that the OpenCL Parboil [53] benchmark has received the optimisation of the CUDA benchmark, potentially making the results deceptive. Rupp et al. [43] made an interesting claim that an optimally configured vector copy operation is representative of the performance of other linear algebra operations.

Using this insight it would seem logical that a well configured vector copy kernel would serve as a suitable low-bias micro-benchmark for exposing achievable peak potential of

a device. This investigation will make use of the STREAM [32] and GPU-STREAM [11] benchmark suites to determine the peak potential of the devices, as they utilise micro-kernels to calculate the memory bandwidth.

5.3.3 *Performance Portability*

Kogge et al. [26] describe performance portability as the "gold standard for evaluating the effectiveness of a programming environment". They suggest that it can be measured by observing the amount of code that must be altered to achieve optimal performance on different architectures. Similarly, Mallinson et al. [31] evaluated performance portability based on the performance penalty from device-specific implementations, but didn't indicate an allowable limit. After evaluating the performance of matrix-multiply kernels Du et al. [39] concluded that the performance was not portable, without any quantification. McIntosh-Smith et al. [34] suggested that if an application can achieve a high fraction of the potential peak performance on all platforms, it can be considered performance portable, but "high" was not quantitatively defined.

The assumption made prior to testing was that no changes are allowed to code between devices, meaning that lines of code cannot be used as a metric. In practice, assessing the performance portability of each of the parallel programming models will be somewhat subjective. By investigating a wide range of models at once it will be possible to provide a rank order, but numerical quantification beyond this will not be provided.

5.3.4 *Conclusions*

The investigation phase of this research project relies upon the collection of consistent and relevant performance data. Ultimately, this data will be collected as an absolute runtime value for each invocation, but other metrics will be used to present this data for observation and analysis. There are a number of experimental constraints that can improve the fairness of a performance investigation and they will be incorporated into this project's experimental design.

Part II

DESIGN AND IMPLEMENTATION

6 IMPLEMENTATION METHODOLOGY

In order to co-ordinate using numerous parallel programming models to develop a range of ports of TeaLeaf, it was essential to devise a solid methodology. This ensured that although the implementation and testing phases were run at the same time, bias was reduced as much as possible and work did not have to be repeated, which would have slowed down the process and reduced the potential outputs.

6.1 VERSION COVERAGE

RAJA and Kokkos were acknowledged as priorities by AWE, to understand their implications in terms of performance portability. The new OpenMP 4.0 and CUDA implementations were chosen to support the evaluation of these models alongside the existing OpenCL implementation. Table 3 demonstrates the current portability of each of the versions:

MODEL	CPUS	GPUS	ACCELERATORS
OpenMP	Yes		Yes (native compilation)
OpenCL	Yes	Yes	Yes
CUDA		Yes	
OpenMP 4.0	Supported	Supported	Yes
Kokkos	Yes	Yes	Yes
RAJA	Yes		

Table 2: Model heterogeneity. 'Supported' suggests no vendor implementations exist, 'Accelerators' covers the Intel Xeon Phi KNC device, and GPUs only includes NVIDIA GPUs.

Each implementation in Table 3 has been chosen to ensure that comparisons can be made between at least three models on the particular device. In particular, the CUDA and OpenMP implementations have been developed to offer highly device-optimised implementations that can expose the best possible performance of the application. All of the tested models have been chosen to enable a balanced investigation and the full set of ports have enough coverage to allow a comprehensive comparative evaluation across all of the devices.

Although OpenMP 4.0 is pitched as a performance portable model, support is currently limited and, while experimental GPU implementations do exist [30], the Intel Xeon Phi KNC is the only officially supported device. In this investigation, the OpenMP 4.0 and OpenCL implementations are used to compare their offloading models to the natively compiled Kokkos and OpenMP implementations.

It can also be noted that while both RAJA and OpenMP 4.0 are reported to be performance portable implementations, they are only officially supported for execution on a single device [54, 23]. This lack of functionally portability greatly influences their current efficacy and is discussed throughout.

6.2 CONSTRAINTS

To ensure that a fair and reliable comparison could be made between the performance portable models, a number of implementation constraints were upheld during the experimental process.

6.2.1 *Limiting Variation*

For consistency when evaluating and comparing the new TeaLeaf ports, great care was taken to ensure that the core computational instructions are highly conserved between each implementation. It must be noted, however, that this did not preclude optimisation outside of the core logic and merely assumed that the problem domain was already solved in an idealistic manner. As suggested by McIntosh-Smith et al. [34], all configuration options, such as internal TeaLeaf-specific parameters and compiler optimisations, were kept as similar as possible between models unless otherwise explicitly stated.

The key components of change between each of the applications are the architectural structuring of the code as well as tuning data handling and traversal patterns. In each case it was necessary to tailor the application code to suit the programming model, with the original application being written in Fortran and the resulting implementations using some mix of Fortran, C and C++.

6.2.2 *Portability Requirements*

Whilst rigorous optimisation was performed for each of the new implementations, all non-portable optimisations were avoided. This constraint ensured that the resulting applications could be run on different vendor devices without significant adjustment, in order to measure the model's performance portability.

6.2.3 *Minimum Required Functionality*

The original application contained a base set of functionality which had to be available in each of the ports to enable complete testing:

- The four iterative solvers: Jacobi, Chebyshev, Conjugate Gradient and Preconditioned Polynomial Conjugate Gradient.
- The original MPI communication conserved, and scalability proven up to 16 nodes.
- Variable mesh sizes, states and precision tuning parameters.
- Original result validation and profiling tools in-tact.

Ensuring that the full feature set was available ensured that none of the ports had improved performance through ignoring aspects of the application, and each of the ports can be released into the UKMAC repository [52] as open source tools for future use.

6.2.4 *Result Validation*

Once the application has iteratively converged upon a solution or reached the maximum permitted number of iterations, it will calculate a high precision checking value. This value was used to ensure that all new ports of the application were able to correctly solve a range of testing problems. The original application had five built-in problems and this project extended that to 25 checking problems for each of the 2D and 3D applications. Those checking problems were created by designing new test cases for the application and calculating the checking values using the existing Fortran implementation.

6.3 TOOLS

The tools used for development were purposefully basic, using the vim text editor for writing the code and the GNU compiler suite for simple testing. The GNU debugger was used for general debugging, and visualisation of data for debugging leveraged the GNU plotting tool. By maintaining this light toolset it was easy to develop on any available machine, including the clustered environments, without significant setup overhead. All code was stored in a private git repository, and each individual port was catalogued by whether it was 2D or 3D etc.

6.4 CONCLUSIONS

Each implementation is only considered complete once all of the available checking problems are successfully passed across multiple nodes using all of the iterative solvers. Developing versions using OpenMP, OpenMP 4.0, CUDA, Kokkos and RAJA, that adhered to the complete set of constraints, will support a reliable and fair comparison of the resulting performance of each of the complementary versions, whilst supporting their long-term success as research tools.

7 DESIGN, PROGRAMMING AND CONFIGURATION

Implementing all of the ports of TeaLeaf was a major component of this project and both 2D and 3D applications were created with the proposed programming models. Each model presented unique challenges and required individual approaches to development. In most cases the original structural code that handled reading and writing configuration files and logs, communication through MPI, and invoking kernels, was conserved. Wherever the mesh data was affected, new kernels were written to support the different models.

Please note that the samples provided are stripped back and more detailed code samples can be found in Appendix B.

7.1 COMPREHENDING TEALEAF

Some up-front investigation was necessary to understand the flow of execution within TeaLeaf, in order to reliably port the code. To facilitate this process, a number of diagrams were drawn whilst manually reviewing the codebase.

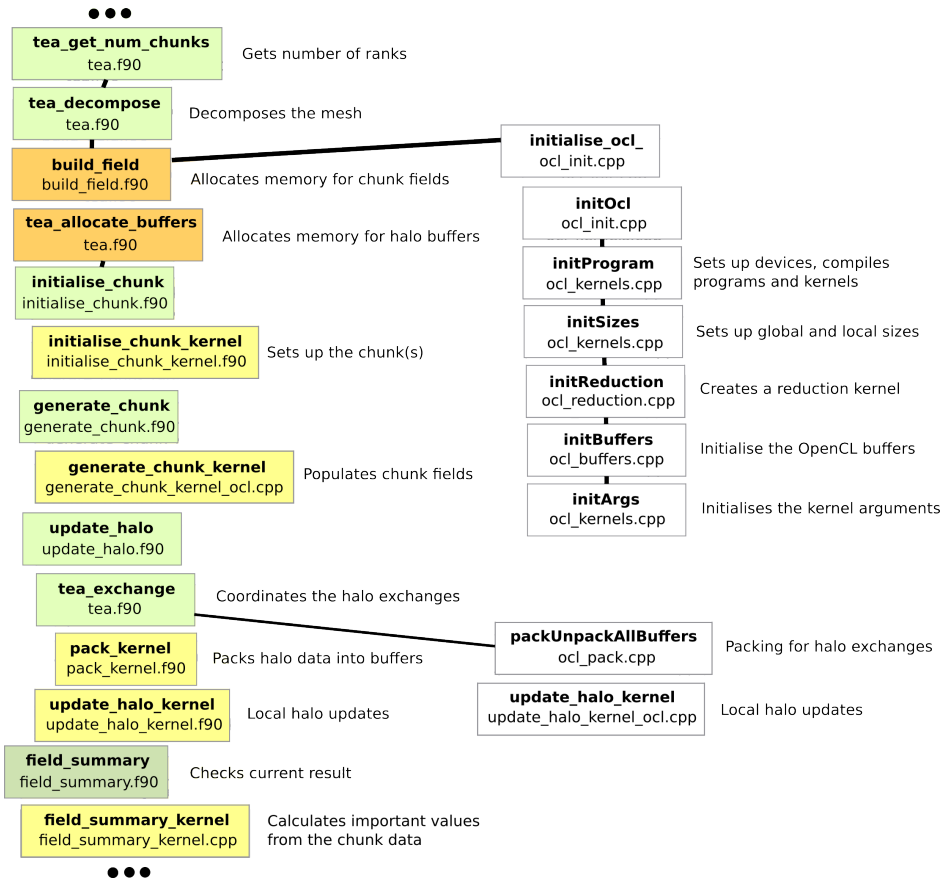


Figure 9: Abridged map of TeaLeaf execution flow, where the green tiles are to be conserved and the other coloured tiles required some level of re-engineering. The white tiles represent the existing OpenCL functions.

The example provided in Figure 9 represents the program flow of the OpenCL implementation of TeaLeaf, the full diagram covers the entire application. By creating these maps it was much easier to isolate exactly which parts of the application would need to be conserved and which would need to be re-written or ported.

7.2 OPENMP C++ AND OPENMP MIC NATIVE

With the entire application mapped and the variable components isolated, it was possible to begin the porting process. In order to facilitate development and testing, a C++ variant of the TeaLeaf application was written and validated against a series of tests. As several of the other models required the use of C or C++, this provided a baseline code that the other programming models could port from, which was not possible from the Fortran implementation. Further to this, the C++ implementation provided an interesting insight into the differences in C++ and Fortan handling of OpenMP, and so became an important application in its own right.

7.2.1 *Design and Implementation*

The design of the port was a full re-write of the application's kernels in C++, to be invoked by the Fortran core application. Each kernel was functionally conserved and OpenMP loop parallelism was added to the outer loops of all of the core loops (Figure 10).

```
#pragma omp parallel for reduction(+: rro)
    for(int jj = HALO_PAD; jj < height - HALO_PAD; ++jj)
    {
        for(int kk = HALO_PAD; kk < width - HALO_PAD; ++kk)
        {
            const int index = jj * width + kk;
            rro[index] += a[index];
            ...
        }
    }
```

Figure 10: Example of OpenMP parallelised loop with reduction.

A flat, one dimensional, structure of arrays (SoA) data layout was maintained, which best suits the memory access patterns that occur in the TeaLeaf solvers [29]. For simplicity, all temporal variables such as counting indices were declared inside the loops to reduce the number of statements required in the OpenMP directives.

7.2.2 *Configuration and Build*

Configuring OpenMP was particularly easy, simply requiring an additional compiler flag -openmp or -fopenmp. Whenever a job was run, the option OMP_NUM_THREADS=<N> could be exported to allow N threads to be run in parallel by the OpenMP runtime. This feature was particularly important when performing the core scaling tests, which required the number of threads to be incremented as part of the test.

In order to run the application on an Intel Xeon Phi KNC, it was possible to compile the whole application using the -mmic native compilation flag, offering an interesting benchmark compared to the other accelerator options.

7.2.3 *Evaluation*

Creating a new C++ serial implementation from the original Fortran code required a significant amount of development but was not particularly complex. Using a baseline implementation made it relatively easy to judge the amount of effort required to take a Fortran application and migrate it to use any of the models. An effort value could be determined by simply adding the time spent on the serial C++ implementation and combining it with the additional effort required for each individual port.

OpenMP is a particularly developer-friendly model and, once the serial C++ implementation of TeaLeaf was implemented, only required a single statement to be added to each outer loop of the application. The additional constructs amounted to roughly 20 additional lines of code, and some changes to the function structure. To avoid having to notify the runtime of private variables in the loop directives, all variables were declared at the deepest possible lexical scope.

7.3 KOKKOS

The Kokkos port demanded a far more significant re-architecture of the application, which involved converting all of the functions containing core logic into computational kernels using C++ templates. The Kokkos framework also requires that all data structures are wrapped into generic types called 'Views', which are templated against the particular device type that the code is compiled with [15].

The View data structures used in this port of TeaLeaf simply wrapped up the original one dimensional arrays. It was considered that some optimisation could be performed by declaring mesh sizes at runtime, using Kokkos to facilitate the optimisation. However, the partners at AWE suggested that for production purposes this would not be of use because of the length of time taken to compile the production applications.

```
template <class Device>
struct CGCalcW
{
    typedef double value_type;
    typedef Device device_type;
    typedef Kokkos::View<double*,Device> KView;

    CGCalcW(TLDims dims, KView w, KView p)
        : dims(dims), w(w), p(p) {}

    KOKKOS_INLINE_FUNCTION
    void operator()(const int index, value_type& pw) const {
        KOKKOS_INDICES;

        if(INDEX_IN_INNER_DOMAIN)
        {
            pw += w[index]*p[index];
            ...
        }
    }

    TLDims dims;
    KView w;
    KView p;
};
```

Figure 11: Example of Kokkos template with reduction.

Each templated kernel had to declare its own storage for the variables it used, which had to be passed in at runtime (Figure 11). The kernels were far more complex than the equivalent OpenMP implementations, but the core logic remained the same. As can be seen in Figure 11, the data access relies upon a single index parameter which is incremented at runtime from 1 to N. The kernels are invoked using a Kokkos loop implementation (Figure 12) and this length is passed in.

```
CGCalcW<DEVICE> kernel(dims, w, p);
Kokkos::parallel_reduce(N, kernel, *pw);
```

Figure 12: Example of Kokkos dispatch parallel reduction.

In each kernel it was necessary to determine the current location in 2D or 3D space from the single index parameter, using modulus operations. Once the row, column (and slice for 3D) were calculated, any index falling outside of the boundary conditions was excluded from the computation. It can be noted that the Kokkos View templates utilised a custom indexing strategy where multi-dimensional indices could be provided as part of an index function call, to support a generic indexing approach [15].

7.3.1 *CUDA and Accelerators*

In order to successfully compile and run the Kokkos port on NVIDIA GPUs and the Intel Xeon Phi KNC, there were certain subtleties that meant changes were required from the original CPU implementation. It must be noted however, that with some simple compiler directives it was possible to create a version that could compile on any of the three supported device types by simply switching a flag at compilation.

Throughout the application there are occasions where the data does not reside permanently on the device, e.g. buffer packing. As the application needed to interface with raw pointers located in the legacy Fortran execution space it was slightly more complicated than if all data was owned by the Kokkos runtime. When compiling for accelerators and CPUs this was not an issue, but for the CUDA implementation it was essential that data ownership was correctly outlined.

In each case constructs had to be written that copied between the device and host using the Kokkos data movement pattern: (1) create a temporal host mirror of the device buffer on the host, to replicate the device layout, (2) copy data into the buffer, and (3) use the built in Kokkos deep_copy command to move the data onto the device. Of course, (2) and (3) are interchangeable depending upon the data flow direction. These operations all proved portable and the Kokkos programming guide [51] describes how the operations are ignored by the runtime unless they are necessary to operate a particular memory space.

7.3.2 *Configuration and Build*

Source level configuration was required to initialise the Kokkos execution spaces correctly. In particular, the CUDA implementation required the application to choose the device to be initialised against, allowing the devices to be split across MPI ranks. Further to this, a single 'device' parameter was held globally in a shared header file, which was changed appropriately when configuring the Kokkos port for particular devices.

The Kokkos library was chain built at compile time, by including the source files local to the project, however this could have been pointed to an external Kokkos installation. It was necessary to provide the targetted device and architecture at compile time, e.g. 'Cuda, Kepler 35' or 'OpenMP, KNC', which directed the template meta-programming, resulting in a device-specific binary.

In order to compile the CUDA implementation the 'nvcc_wrapper', a Kokkos-specific script, was utilised to generate sane nvcc compilation instructions. This required at least CUDA 6.5 and Intel compiler version 14.0 in order to run successfully [51].

When compiling Kokkos to run on accelerators the only change necessary was to add the -mmic compiler switch to all of the legacy architectural code, which forced it to be compiled in native mode.

7.3.3 *Evaluation*

The development phase of this application port demonstrated that successfully implementing Kokkos into an application is easiest if (a) the application is already written in C++, where a pervasive use of templates would likely expedite the process, or (b) the application is written in another language and the developer is comfortable with conversion to and use of C++ templates. As Kokkos has been written to emulate characteristics of standard template library features, it is quite intuitive if this style is familiar.

In order to completely re-develop a large scale application to use Kokkos, the burden may be overwhelming if the original application was written in a language other than C++ or perhaps C. Kokkos requires little boilerplate code, a slight advantage over OpenCL, which means that the development effort is spent purely on creating the functors and then organising data transfer and execution dispatch.

Because the kernels receive only a single index parameter, it is up to the developer to re-calculate indices, which could definitely be improved in the implementation. Once the basic format of the kernels is understood they all look fairly similar, but they require more lines of code than equivalent CUDA or OpenCL. The frameworks handling of reductions is very clean and intuitive, handling multiple parameters through defining a simple data structure.

One limitation of using TeaLeaf to evaluate Kokkos is that the data structures are very simple one-dimensional arrays. As Kokkos is so data structured oriented this has more than likely excluded some useful features from this study, which may make Kokkos more appealing for other applications.

7.4 RAJA

The RAJA implementation was provided by Lawrence Livermore National Labs, but they are currently *in development* and were only released privately to allow this project to test the concept. The design for the port required all loops to be wrapped up into one of the built-in RAJA loop constructs, either with or without a reduction (Figure 13).

By providing a simple 'policy' parameter it is possible to alter the loop dispatch implementation by changing this parameter on a device, although this currently only works for OpenMP [23]. In the RAJA source there are some placeholders for SIMD device implementations, which suggests that this functionality will be available in the future.

```
forall<policy>(all, [&] (int index) {
    p[index] = a[index]*a[index];
});

forall_sum<policy>(inner, [&] (int index, double* rro) {
    *rro += a[index]*p[index];
});
```

Figure 13: RAJA standard parallel for loop (top), with reduction (bottom).

The 'all' and 'inner' parameters provided to the loop represent array traversal indirection lists that are pre-computed during application initialisation. In order to handle the ghost cells surrounding the problem domain, all indices were stored inside RAJA List Sets, which allow non-contiguous access to be precomputed that excludes the ghost cells.

It was necessary to create two new implementations of the RAJA forall loops: (1) the halo updates which took two precomputed indices, and (2) the buffer packing for MPI communications, which required both a domain index and a counting index to read or fill the buffer. In the end, many of the functions within TeaLeaf utilised custom implementations of RAJA *forall* loops created to suit the application, which the developers openly encourage and suggest sets RAJA aside from other programming models [23].

7.4.1 *Configuration and Build*

The port was configured to chain compile the RAJA source code, which was held locally alongside the application. Further, it was necessary to compile the application as C++11, using the -std=c++11 compiler flag. Although RAJA was only utilised for CPUs in this research project, the global policy could be changed to alter the RAJA implementation at compile time to take advantage of particular devices.

7.4.2 *Evaluation*

The bulk of the development of RAJA closely represented more development effort than OpenMP but less than Kokkos, and did not require large architectural revisions. As all arrays were accessed using the special pre-computed indirection list, the resulting logic was simplified. Additional effort was required because the indirections lists had to be created during mesh generation, and actually lead to all of the loops being separated from the logic.

If all of the loops are identical within an application, then there is a significant reduction of duplication, but this is not even the case for TeaLeaf, which has a single specific focus. It would be particularly important for a large application to carefully manage the generation of these indirection arrays to avoid an explosion of code, and to encapsulate the logic and indirection together if possible.

In order to successfully complete the port, new implementations were required to handle reductions with multiple parameters. Given that TeaLeaf is a small application it would not be surprising if there were even more requirements for a production application, and this would represent a large overhead if the work had to be repeated for multiple devices as well.

It must be noted that changing each functional loop to utilise RAJA was at least as much effort as adding OpenMP statements, and the current lack of GPU support renders the memory improvements limited. What the RAJA concept does really achieve is clean code using the C++11 lambda feature, and an interesting approach to indirection for complicated memory access patterns. In TeaLeaf's case the memory access pattern is fairly flat, only requiring exclusion of the ghost cells, and so TeaLeaf did not particularly benefit from the indirection.

7.5 CUDA

To port the original OpenMP implementation to CUDA, kernels needed to be created for all of the core functions, and some additional architectural code written to support data allocation and the copying of data to and from the device with synchronisation. A 2D implementation of TeaLeaf CUDA was already written prior to this research project, but to ensure consistency of optimisation and to limit variation, a new 2D CUDA implementation was developed from the new 3D port and used in all testing.

```
__global__ void CuKnlCGCalcW(
        const double* p,
        double* pw)
{
    __shared__ double pwShared[BLOCK_SIZE];
    pwShared[threadIdx.x] = 0.0;

    const int gid = threadIdx.x + blockIdx.x * blockDim.x;

    CUDA_INDICES

    if(INDEX_IN_INNER_DOMAIN)
    {
        pwShared[threadIdx.x] = p[gid]*p[gid];
    }

    Reduce<double, BLOCK_SIZE/2>::Run(pwShared, pw, SUM);
}
```

Figure 14: Imitation CUDA kernel, real code samples available in Appendix A.

CUDA kernels Figure 14 require that pointers to all device memory are passed in from the host code, and simple data types are stored in registers or constant memory [9]. All data stored on the device was allocated during application initialisation using the built in CUDA malloc operations. When there were buffers that needed to be copied to/from the device, the built in CUDA memcpy and synchronise functions were utilised.

7.5.1 *Reductions*

Efficient implementation of reduction operations was more involved for CUDA than the other ports, as there are no built in options, and GPU reductions are more complicated. A naive implementation would simply have all threads write the value into global memory, but this would be incredibly inefficient given the bandwidth to global memory.

To effectively take advantage of a GPU, it is best to assign each thread an item in the reductee array to be reduced and first reduce it into shared memory as a binary reduction Figure 15. Note in Figure 14 that to improve readability, a template meta-programming

trick is leveraged from the original 2D implementation. It expands at compile time into an NVIDIA optimised loop that reduces all values within a thread group in local memory and writes the final value back to another global array upon completion.

After the kernel has been completed, a global array contains a value for every group of threads, which is then reduced independently by invoking a number of additional reduction kernels.

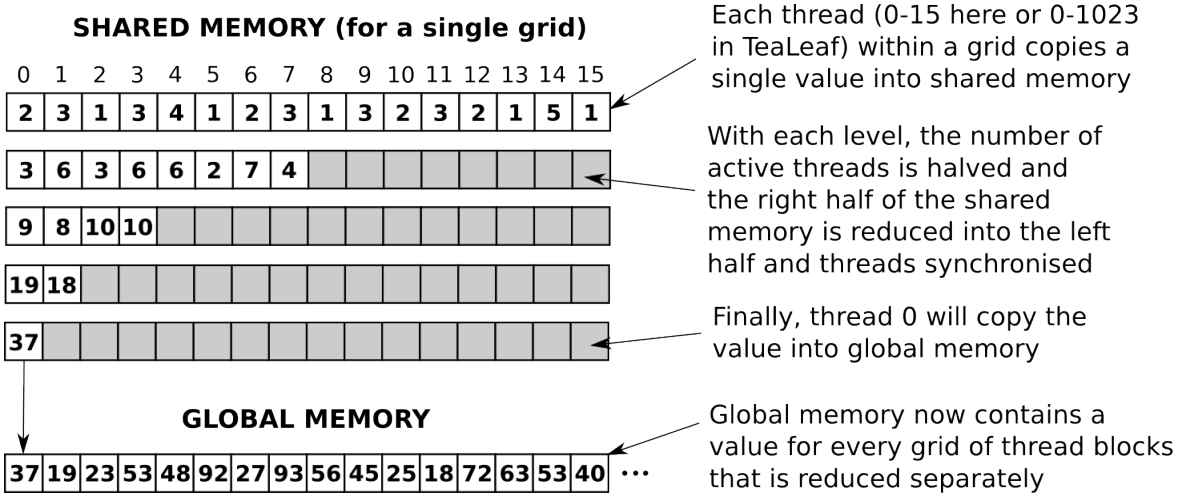


Figure 15: CUDA reduction design, grey boxes represent inactive threads.

7.5.2 *Configuration and Build*

Configuring CUDA required declaring the architecture and compute capability of the device(s) being compiled on, using a command like 'arch=compute_35,code=sm_35'. The compilation is performed by the NVIDIA 'nvcc' compiler, which requires all standard compilation flags to be passed through the '-Xcompiler' parameter. In the case of TeaLeaf, the 'nvcc' compiler handled compiling the .cu files and the standard MPI compilation wrapper scripts were used for the C++ and Fortran architectural code.

7.5.3 *Evaluation*

CUDA was specifically chosen as a TeaLeaf port to provide a basis for evaluation against the performance portable models, but also provided an interesting insight into the complexity inherent with creating device-specific implementations. The port was more complicated to develop than OpenMP, RAJA and Kokkos, and required significantly more specialist knowledge, including a good understanding of the memory hierarchy on a GPU. Once an efficient reduction has been developed, it will be applicable to any function using reductions in any application, so the expended effort is diminished over time. Other than the reduction and some memory allocation, the model is quite light on boilerplate code, in contrast to OpenCL.

The kernels that CUDA uses have a fairly simple C-based syntax, only adding a handful of directives, mostly focussed upon data storage. When dispatching a kernel it is possible to decompose the problem across a grid of threads, but for TeaLeaf it was simpler to use a one dimensional index like Kokkos and calculate the offsets from the ghost cells. In

contrast, OpenCL allows you to specify an initial offset when dispatching a kernel which makes handling ghost cells easier.

7.6 OPENMP 4.0

Out of all the ports, OpenMP 4.0 was the only one that required multiple architectural revisions during the optimisisation process. This meant that the original design does not represent the final implementation at all. As opposed to describing the deficient design, this section pre-emptively describes the final optimised design and the other designs are briefly described in subsection 9.3.2.

The conserved parts of the TeaLeaf application contain a particular module that controls each step of any solver (running many internal iterations), which is referred to as the controlling code. Firstly, all of the controlling code had to be changed to no longer utilise pointers that were members of a structure, so all pointers were passed through a function call and stored locally to the function. Most of the controlling code was wrapped as a **target** region with data persistence through the **target data** command, as seen in the top of Figure 16.

tea_solve.f90

```
! Declares the region as a target, updates global variables and maps local variables
!$omp declare target(ext_cg_calc_ur, ext_cg_calc_p, ...)
!$omp target update to(max_iters, ...)
!$omp target map(tofrom: p, u, r, ...)

do n = 1, max_iters
    call ext_cg_calc_ur(u, r, ...)
    call ext_cg_calc_p(p, ...)
enddo

!$omp end target
```

ext_cg_kernel.c

```
// Declares the functions as callable from an offloaded target region
#pragma omp declare target

void ext_cg_calc_ur_(double* u, double* r, ...);
void ext_cg_calc_p_(double* p, ...);

#pragma omp end declare target
```

Figure 16: The final design of the OpenMP 4.0 implementation.

All global variables and functions had to be declared as targets, with particular commands to update the global variables at certain times during runtime. The applications were written so that they could handle C and Fortran kernels with OpenMP 4.0 offloading, but both took a slightly different approach, where C used the syntax in the bottom of Figure 16 and Fortran required a single line to be added to each offloadable function. Further to this, the individual C function calls had to be explicitly declared as external targets in the controlling code, whereas the Fortran modules were automatically discovered.

7.6.1 *Configuration and Build*

The configuration for OpenMP 4.0 was identical to the configuration for OpenMP, simply adding -openmp for Intel and -fopenmp for GNU compilers.

7.6.2 *Evaluation*

In order to successfully embed OpenMP 4.0 into the application it was more complicated than OpenMP, requiring numerous additional declarations and some architectural alterations. Pre-empting the best way to incorporate the OpenMP 4.0 features into the application proved impossible because the discovered research focusses upon small examples that can't represent the architectural change required for a legacy codebase or non-trivial application like TeaLeaf [30, 38, 54].

An important discovery was that data sharing is only possible within lexically structured scope [4]. This works well for trivial examples and well structured applications like TeaLeaf, but more than likely there would be many other applications that would significantly suffer from this restriction.

If the final design had been known up-front, it would have been reasonably straightforward to implement the necessary changes but, unlike the other applications, the implementation is currently restricted to a single node. It is likely that the effort required to copy data to and from the device is similar to updating global variables and so it is not anticipated to present a large overhead. Ultimately, those who have used OpenMP to good effect will find the syntax familiar but may have to contend with more complicated architectural decisions than they would anticipate.

7.7 OPENCL

Whilst the 3D OpenCL implementation was already created, issues with the 2D OpenCL implementation meant that a new 2D port needed to be created for this research project from the 3D version. Also, there were small configurational changes required to make the OpenCL implementation work on the CPU and accelerator cards. In particular, to compile for the CPU and accelerator, it was necessary to change settings in the TeaLeaf configuration file to target the correct devices.

7.7.1 *Evaluation*

Although the OpenCL port was not actually developed as part of this research project, it is clear that it is one of the most complicated programming models from the selection.

The work required to convert the 3D implementation into a 2D version demonstrated that OpenCL requires specialist knowledge to support its functional portability, which wasn't necessary for the other models. As with most models, once the boilerplate code is finalised, creating new kernels and adding them to the execution pipeline becomes easier and the syntax is very similar to CUDA.

It is expected that, of all of the models presented, the OpenCL implementation would likely require the most significant development cost if porting a legacy codebase. It must be noted that there is a trade-off between development complexity and the fact that OpenCL is an open standard that is implemented by most of the top vendors. In fact, OpenCL is

```
__kernel void tea_leaf_cg_solve_calc_w(
        __global double* p,
        __global double* pw)
{
    __local double pw_shared[BLOCK_SZ];
    pw_shared[lid] = 0.0;

    __kernel_indexes;
    const int gid = column + row*(x_max + 4);

    if (row <= (y_max + 1) && column <= (x_max + 1))
    {
        pw_shared[lid] = p[index]*p[index];
    }

    REDUCTION(pw_shared, pw, SUM);
}
```

Figure 17: Imitation OpenCL kernel, real examples available in Appendix A.

far more functionally portable than any of the other implementations investigated in this research, targetting multiple vendors and even devices such as field-programmable gate arrays (FPGAs) [31].

7.8 CONCLUSIONS

Having only had experience with C, C++ and OpenMP, programming ports with such an extensive range of models represented a significant challenge that was overcome by drawing upon the limited available research and approaching the development with a methodical and scientific approach.

By implementing the ports with a fresh perspective, it is anticipated that the insights presented are both unique and unbiased. With the full complement of applications successfully implemented and configured, so that they conformed to the constraints outlined in section 6.2, the testing could be performed with confidence regarding the experimental validity. Also, the developed ports greatly extend the TeaLeaf benchmarking suite to support future research.

8 EXPERIMENTAL DESIGN AND PROCESS

In order to efficiently and reliably collect performance metrics for the newly implemented ports, it was essential to develop and maintain a strict test process up-front.

8.1 PLATFORMS AND DEVICES

Choosing representative devices has been important for evaluating the ability for each parallel programming model to perform well on modern supercomputing platforms. The top 2 supercomputers of the supercomputing top 500 list (June 2015), contain the NVIDIA K20x, Intel Xeon Phi KNC and Intel Xeon E5-2692. Given that very similar devices are available on the Blue Crystal phase 3 (BCp3) supercomputer, the performance testing can provide reasonable confidence in the efficacy of each model.

As TeaLeaf is a memory bandwidth bound problem, it is quite important to acknowledge the difference between the vendor reported bandwidth and the proven peak bandwidth. In order to assess this gap, the STREAM (Triad) [32] and GPU-STREAM (CUDA) [11] benchmarks were setup and run on each device on Blue Crystal phase 3, and the results are provided in Table 3. It is believed that directly calculating the bandwidth provides a fair representation of the maximum potential performance for each individual device *and* configuration.

DEVICE	CORES	VENDOR M.B.	STREAM M.B.
Intel Xeon E5-2670 CPU x 2	16	51.2 GB/s (each)	76.2 GB/s (both)
NVIDIA Tesla K20 GPU	2496	208 GB/s	150.6 GB/s
Intel Xeon Phi KNC	60	320 GB/s	159.9 GB/s

Table 3: Statistics of tested devices, where M.B. is memory bandwidth.

For the CPUs, the STREAM benchmark was altered to compile with the Intel compiler, additionally setting the KMP_AFFINITY parameter to "compact", which forced the correct utilisation of both processors and achieved the best results. When running on the Intel Xeon Phi KNC, it was necessary to recompile as a native application as discussed in section 7.2, copy the STREAM benchmark to the device, and configure it with parameters discussed by Raman et al. [40]. The GPU-STREAM, developed by the HPC Group at the University of Bristol, required no additional configuration for the NVIDIA K20 GPU.

On a single node of Blue Crystal, a pair of Intel Xeon E5-2670 CPUs are available, both of which are utilised in the performance testing included in this research. There are also special nodes available that contain pairs of NVIDIA K20 GPUs or Intel Xeon Phi KNC accelerators, but note that while the testing secured the entire node for a test, only a single device was ever used during a particular test. The reason for this decision is that the paired CPUs work in a shared memory environment, whereas the GPUs and accelerators have their own independent memory and must be considered separate devices. By excluding

non-shared memory environments and technologies such as MPI, it is possible to focus the performance evaluation directly on the programming models themselves.

8.1.1 *Conclusions*

Three modern HPC devices have been selected for testing that are representative of the HPC devices in many of the top supercomputers in the world. Assessing the ability for each of the application ports to take advantage of such heterogeneous devices will truly expose the efficacy of their respective models, in order to understand the potential benefits of the performance portable programming models over their device-specific counterparts.

8.2 TEST AUTOMATION

Test automation ensured that time was not wasted on the repetetive aspects of collecting the performance data, but the priority was always to collect the best quality data possible. The test handlers were designed as encapsulated modules to have a minimal impact on the applications, meaning the suite of ports did not have to undergo significant repetetive development. When working with such a large number of applications at once, it is essential to mitigate the risk of introducing hidden measurement bugs.

8.2.1 *Queuing*

Multiple PBS job submission scripts were created, an example of which can be found in Appendix B. A single script was created for each combination of application and test, allowing the tests to be repetetively queued without variation. In many cases the script would run the application multiple times per solver, collecting data for each run.

There were limitations with the BCp3 queue system that shaped the testing process, for example, jobs for the Intel Xeon Phi KNC nodes could only be queued for a maximum of 1 hour. When running the application with large mesh sizes, some of the solvers required over 1 hour for a single solve, making testing them impossible.

Further to this, the nodes available on the supercomputer were often very busy, and gaining access to individual nodes meant competing with other researchers for the resource. In total, the application was run over 15,000 times, adding up to more than 450 hours of wallclock time across all tests, and much more if initialisation and teardown time could be accounted for. In order to fulfil this it was necessary to setup and prepare tests to be scheduled and run overnight when the supercomputer had less activity.

8.2.2 *Performance Logging*

To manage the volume of data, it was essential that data collection did not rely upon copying results from file to file. A performance logging component was written in C++, a sample of which is presented in Appendix B, which served as a simple log-file analyser for the TeaLeaf application. At the end of each test run the application would call an external function that archived the TeaLeaf output file and created a new CSV entry for the performance run in the designated test suite performance log file.

Importantly, each run was validated using the internal TeaLeaf validation test and logged, which meant that any faulty runs were excluded from the final calculations.

```
tealeaf_kokkos, 1, 16, 316x316, cg, PASSED, 0.569955825806
tealeaf_openmp, 1, 16, 316x316, cg, PASSED, 0.330569982529
tealeaf_raja,   1, 16, 316x316, cg, PASSED, 0.394473075867
```

Figure 18: Example of simplified log file entries (Application, Number of nodes, Number of threads, Problem size, Solver used, Success, Wallclock time).

Storing the data in this manner has kept an auditable track of all of the performance results since the project started. This has allowed repeated checks of the validity of the data and multiple visualisations to evolve from the same data sets.

8.2.3 *Performance Visualisation*

Given the volume of data, it was not possible to analyse it in raw form and quickly observe problems or points of interest that could inform additional testing or optimisations. To overcome this a set of python scripts were created that could generate graphs by reading in the performance log files. These allowed continual feedback and sanity checking of the results, and were later used for the final visualisations for this report.

In order to deal with duplicate runs, generally five or more runs for each application and test suite, the results were collected together and the minimum result taken. The reasoning for this choice is that the fastest possible time achieved by an application is the true reflection of its optimum performance, any increase in time is likely caused by unknown external influences. If instead the calculations were to be based upon averages, this could inaccurately represent the performance when faulty runs occur, which has been observed several times during this project.

8.2.4 *Conclusions*

Once the test automation was perfected, it allowed the full suite of tests to be run for all applications with one command, allowing a wider range of tests to be designed and used for the process. Although the initial cost of developing the test automation tools was quite significant, perhaps 8 days net development effort, the resulting gains far outweighed this in the long term. Sample codes for logging, visualisation and profiling are available in Appendix A.

8.3 PROFILING

There is a range of existing tools available for profiling applications, but each comes with its own standards and approaches, making it potentially complicated to manage profilers for so many different applications and devices. The TeaLeaf problem has been chosen specifically because the algorithms required by the solvers are reasonably simple and compact, meaning that the application lends itself well to custom profiling.

To handle profiling, a generic module was developed that was used to surround all of the core function calls in each application, a sample of which is available in Appendix B. This utilised high performance timing counters to determine the runtime profile of every function, and allowed the results to be printed with each test run. C macro statements were used to force the compiler to remove all profiling calls, using a compiler directive, unless

profiling was explicitly enabled, which ensured that the profiling suite did not affect the wallclock measurements taken when measuring performance during the actual test runs.

8.3.1 *Bandwidth*

As demonstrated in section 8.1, each device has a limited bandwidth for memory access to DRAM. Given that the TeaLeaf problem is memory bandwidth bound, this metric is particularly important for understanding the achieved level of optimisation on a particular device. Measuring the bandwidth of each application has been a non-trivial task, because each architecture handles memory access uniquely. The difficulty lies in understanding how best to calculate the bytes read and written in a manner that provides a fair experimental result, before plugging into the standard equation [8].

$$\frac{\text{Num Calls} \times (\text{Bytes Read} + \text{Bytes Written})}{\text{Program Runtime} \times 1024^3}$$

For instance, the CUDA implementation of TeaLeaf was profiled with the CG solver on for the 2048×2048 mesh size. The bandwidth was measured using a simple calculation of the bytes read and written by each function (Figure 19).

```
w[index] = (1.0
        + (Kx[ii + 1] + Kx[ii])
        + (Ky[ii + width] + Ky[ii])) * a[ii]
        - (Kx[ii + 1] * a[ii + 1] + Kx[ii] * a[ii - 1])
        - (Ky[ii + width] * a[ii + width] + Ky[ii] * a[ii - width]);
```

Figure 19: Calculate W function of CG solver.

Each time an array access is invoked (variable name followed by square brackets), there will be a call to some part of memory to fetch this data. Naively counting the number of array accesses would come to 14 in total, but this would overcount the number of actual memory access occasions because redundant array access will be optimised away by the compiler. In this function that leaves 9 unique pieces of data that are read from memory and 1 which is written.

This naive method of counting reads and writes was used to determine the bandwidth achieved by the CUDA implementation on an NVIDIA K20. However, the result for this particular function was 327 GB/s, an immediately alarming result given that the GPU-STREAM benchmark only achieved 150.6 GB/s.

The cause of the incorrect bandwidth calculation appears to be contiguous memory access. In particular, access to DRAM is often a very high latency and bandwidth limited resource, but low latency cache is utilised by the hardware to reduce the amount of repetetive local or contiguous memory access. CPUs and GPUs will cache whole lines of memory, e.g. 128 bytes of contiguous memory, and array accesses like Kx[index] and Kx[index + 1], which reside in neighbouring memory banks, will be cached into a fast cache memory. The fast cache memory is limited but has far higher bandwidth and lower latency than DRAM.

Collaboration with the UoB HPC group resulted in two opposing opinions regarding this point: (1) count close contiguous access as free, a loose upper bound, (2) consider caching as perfect and only count the size in bytes of all buffers accessed, which is essentially a lower bound. Both of the options are used in later testing (section 10.7).

8.3.2 *Reviewing and Reports*

Where necessary, tools such as the Intel Vectorisation Reports feature of the Intel compiler [31] were used to determine whether the compiler was leading to correct code for a particular port. In some exceptional cases, it was useful to extend the investigation beyond this and look deeper into the code generated. This essentially required disassembling a port's binary file into assembly and directly reviewing critical sections of the code to reveal problems in the resulting application.

8.3.3 *Conclusions*

Profiling is an essential step in the optimisation process, to ensure a focussed and targetted approach to improving the performance. However, it is also incredibly useful for exposing issues that cannot be optimised, and there are many scenarios where this research project uses fine-grained profiling results to make suggestions about potential causes of poor performance.

8.4 TEST DESIGN

The test suite was designed to gain an improved understanding of the TeaLeaf application and the chosen parallel models. The tests attempted to alter a single parameter or analyse a particular configuration, to investigate the performance of each of the models under different conditions.

8.4.1 *Configuration*

In order to adapt the parameters for each test run, the TeaLeaf configuration file was leveraged, with hundreds of individual configuration files being created to cover each case. By using the configuration file it was possible to run different test suites without changing the code to accept new parameters, significantly reducing the risk of affecting the results with bugs.

Further to this, the variation between applications was greatly minimised as all applications utilised the same configuration files and as such identical parameter profiles. This meant that there were a substantial number of configuration files to manage but all tests could be traced back to the precise configuration file utilised for test validation.

8.4.2 *Test Details*

None of the parallel programming models evaluated by the research projects offered support for communication outside of shared memory environments. A byproduct of this is that inter-node testing offered no additional insight into the performance portability of the applications and so the testing was limited to single node tests. Strong scaling tests were run up to 16 nodes for all applications except for OpenMP 4.0, to prove that the ports were free from bugs, but the performance of the strong scaling tests was not relevant to the project.

A number of tests were created, each intending to expose a different application performance characteristic:

- **Chebyshev Presteps**: An independent parameter that controls the number of Chebyshev steps run prior to the Conjugate Gradient solver. Tested setting the parameter between 1 and 100 for all applications, to find optimal setting.
- **PPCG Inner Steps**: An independent parameter that controls the number of PPCG steps run during an iteration. Set the parameter from 10 to 600 across a range of mesh sizes, for all applications, to find optimal setting.
- **Even step**: Increased the mesh size from $100,000$ to $1,500,000$ (on a single node) in steps of $100,000$, to compare the performance scaling of each application.
- **Power of 2**: Scaled the mesh size in powers of 2, from 32×32 up to 4096×4096, attempting to further understand the performance profile of the application.
- **Core Scaling**: For the OpenMP-based CPU ports, this test increased the core count from 1 up to 16 to understand the performance scaling possible with those models.
- **Single Step**: This test incremented the mesh size by one cell each run to look for patterns in preferred mesh sizes. The results are excluded for brevity as the performance for such granular steps was chaotic and offered no interesting information.
- **GNU and Intel Compilers**: This test compares the achieved performance of relevant ports when compiled with either the GNU or Intel compilers.
- **Test Run Variability**: For Kokkos and OpenCL, many test runs were performed to observe the variability experienced over a number of solves.

Each of the tests was run at least five times for all versions of the applications including, where appropriate, configurations of each application on heterogeneous devices. Each test was run for the three principal solvers: CG, Chebyshev and PPCG, with the Jacobi being excluded because it presents a sub-optimal solution for the problem domain in all cases. It may also be noted that a further repetition existed between 2D and 3D implementations of the applications, but only the 2D results are presented because of space limitations.

The largest mesh size used was 4096×4096, because it is the point of mesh convergence, and going beyond this size offers no additional scientific information in the result.

8.4.3 *Conclusions*

The designed set of tests have been conceived to attempt to isolate the performance of each of the models while restricted to a single node and device. Altering parameters, configuration setttings, and the problem being solved makes it possible to collect enough varied data to contrast each of the models and understand their performance profiles for each device.

Part III

RESULTS, ANALYSIS AND CONCLUSIONS

9 OPTIMISATIONS

The optimisation process involved a test driven and iterative adjustment of the application code and configuration. This analysis is presented as part of the results section because it represents many of the intermediate results collected through the project process. In order to provide a fair comparison between the performance portable and device-specific models, optimisation was performed wherever possible without altering the problem solving logic or affecting portability.

9.1 TEALEAF PARAMETER TUNING

An analysis was performed on the TeaLeaf application to determine what parameters were available for tuning. This uncovered two application-specific parameters that directly affect the route that two solvers take to find a solution to the heat conduction problem presented. It can be noted that because they both affect different solvers, the parameters are independent of each other.

9.1.1 *Chebyshev Pre Steps*

The Chebyshev pre steps parameter determines how many iterations of the CG solver will be run prior to the Chebyshev Solver being invoked. This allows eigenvalues to be approximated and the default for the application was 20.

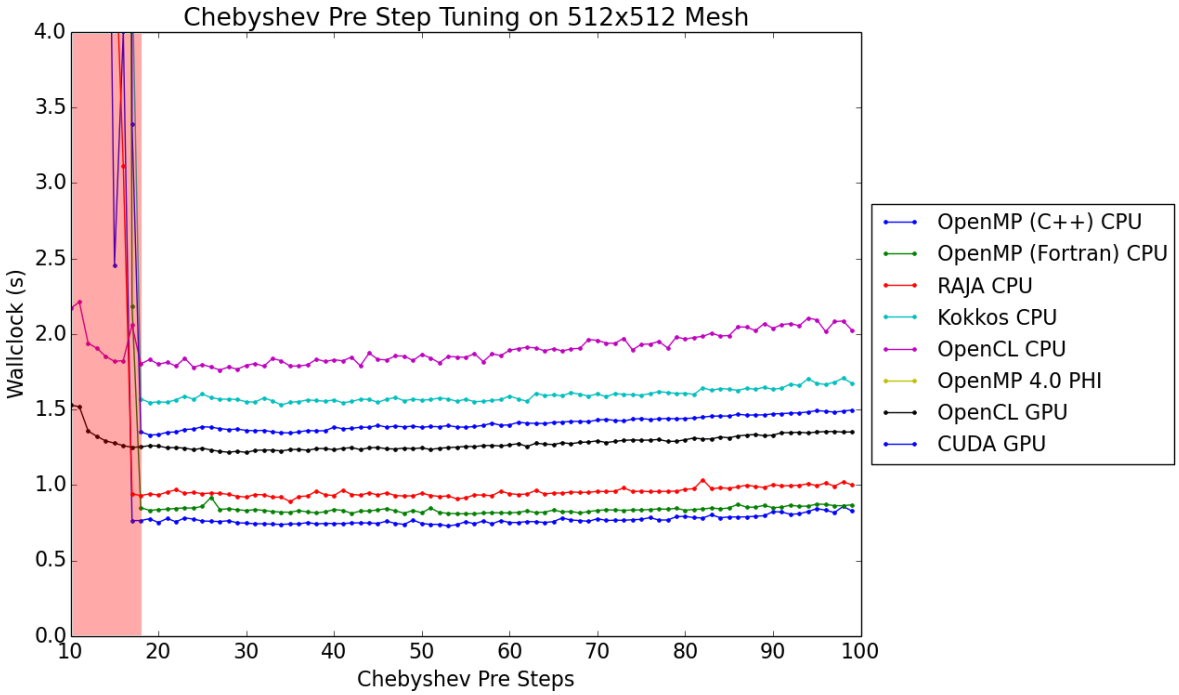


Figure 20: Chebyshev Presteps Tuning for 512x512 mesh size.

The results of the parameter tuning exercise are presented in Figure 20, and the section highlighted in red demarcates a set of values where the models failed intermittently. It is believed that this occurs because running too few CG steps leads to an inaccurate approximation of the eigenvalues. Aside from this, it appears that the best value for the Chebyshev pre steps setting is around 20, as the performance generally decreased as the number of steps increases past this point.

As the parameter increases, the portion of the solve time handled by the CG solver increases, and so the OpenCL CPU implementation suffers a worse performance decrease over time because it handles the CG solver less efficiently than the Chebyshev solver. Further testing found that the parameter choice holds even for larger mesh sizes, and it was discovered that the code will ignore the pre steps value if the solution error has not decreased past a hard-coded boundary value. This information lead to the parameter being set at 20 for all performance testing.

9.1.2 *PPCG Inner Steps*

Preliminary testing demonstrated that the PPCG solver was around 300% slower for some mesh sizes than the other solvers. This performance issue was caused by the PPCG inner steps parameter, which controls how many inner steps of the PPCG solver run per iteration. It was set to 20 in the original TeaLeaf input file, which had good performance for smaller mesh sizes, but poor performance relative to the other solvers as the mesh size increased.

Optimal settings were searched for three major mesh sizes 512×512, 2048×2048 and 4096×4096.

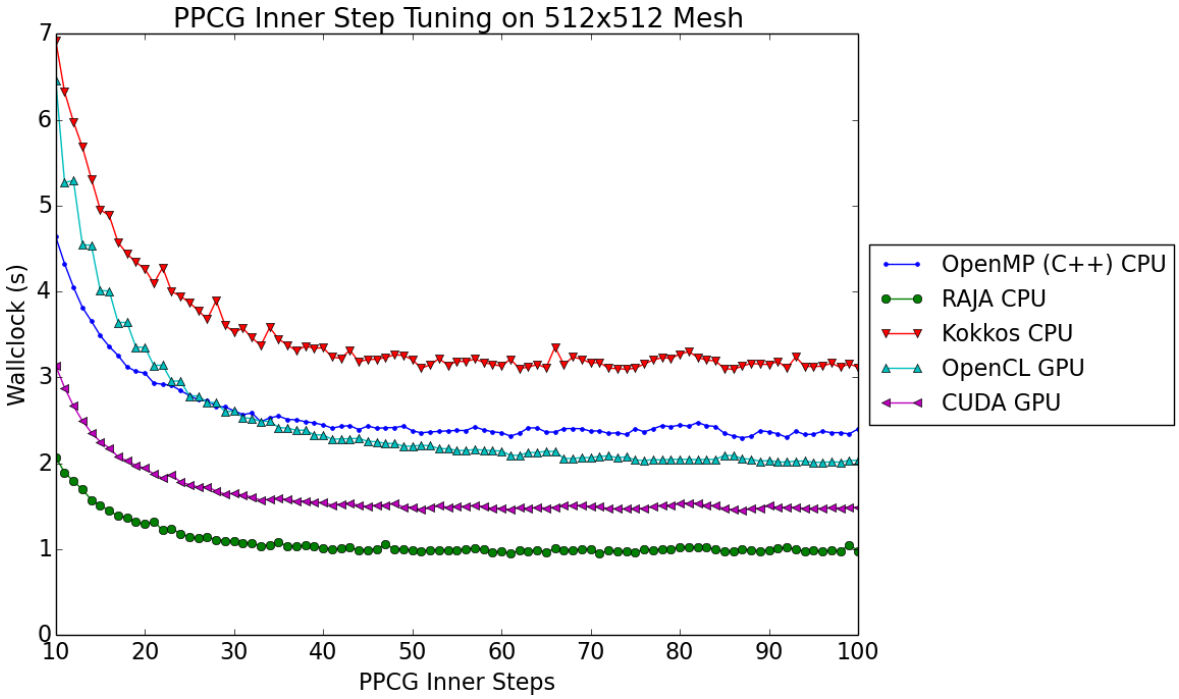


Figure 21: PPCG inner steps for mesh size 512×512.

In the 512×512 case (Figure 21), the best parameter can be taken as between 50 and 100 (potentially higher).

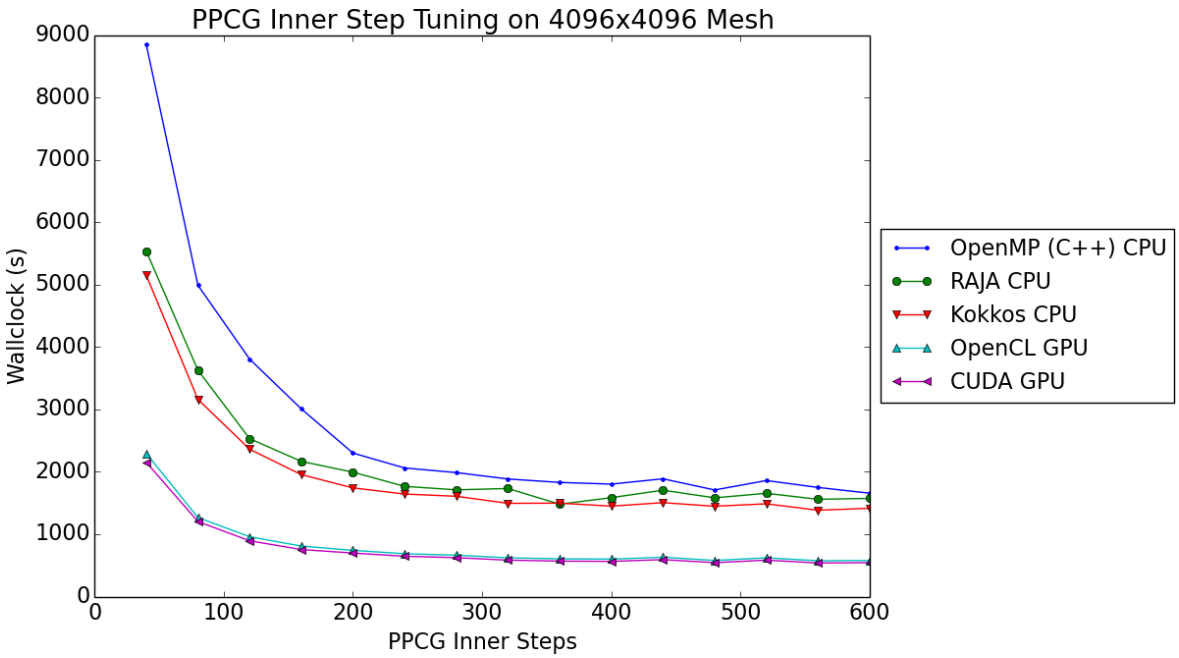


Figure 22: PPCG inner steps for mesh size 4096×4096.

Comparing both the 512×512 case in Figure 21 and the 4096×4096 case in Figure 22 demonstrates that when the mesh size is increased, the lowest optimal parameter setting will be a higher value. The collected results suggest that the ideal setting for 512 is at least 50, 2048 is at least 150 and 4096 is at least 350. In all cases the models exhibit identical best fit values, this is because the parameter only affects the number of steps performed by the solver and does not influence memory or other model-affecting properties of the problem.

9.1.3 *Conclusions*

This tuning exercise determined optimal values for two key parameters in the TeaLeaf application, which were known to be independent of each other and all of the model-specific parameters, and have been shown to be invariant between models. While these parameters were the only ones that affected all of the models at once, there were other model-specific parameters that are dealt with in section 9.3.

9.2 DATA ACCESS

All data buffers were held as single dimensional arrays of double precision values in the original application and this structure was maintained as research has suggested it will prove optimal for the devices chosen [29, 18]. The TeaLeaf algorithm is already known to represent a memory bandwidth bound problem, and so optimisation generally focussed upon minimising expensive data transfer [55]. In each port the indexing was carefully organised to minimise the amount of data access that occurred. This meant that there was some additional computational complexity required to handle data access, especially because optimal data access ignores the ghost cell region (the red cells in Figure 23).

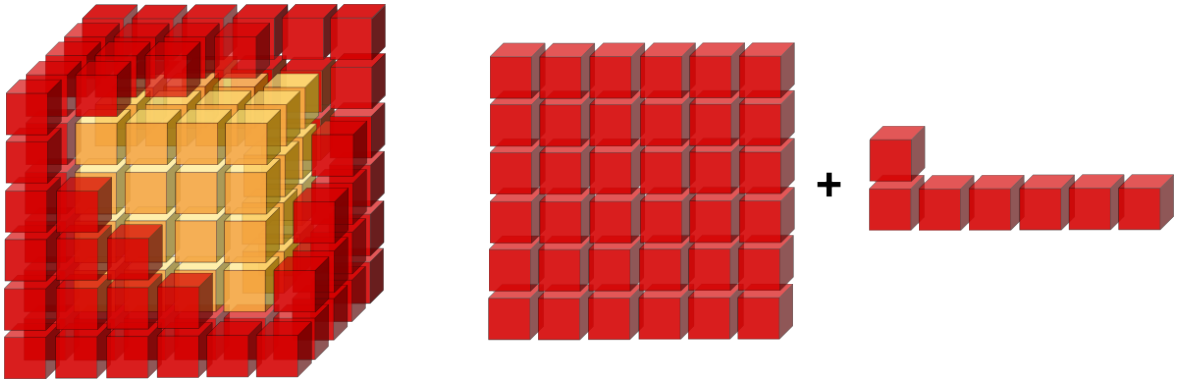


Figure 23: Problem mesh (left), offset required to reach first cell (right).

For the OpenMP (and 4.0) implementation this can be handled quite simply with double or triple loops that exclude the outer cells. RAJA similarly allowed multiple loops to be used to create a set of indices that were used in the RAJA indirection index access strategy.

Kokkos and CUDA required a different approach, as the computational kernels only have access to a single 1D index parameter that must be used to calculate location within 2D or 3D space. After calculating the row, column and potentially slice using modulus operations, two different strategies were performance tested in both ports: (1) use conditional statements to ignore the ghost cells, and (2) calculate offsets based on the current location in space (Figure 23 right).

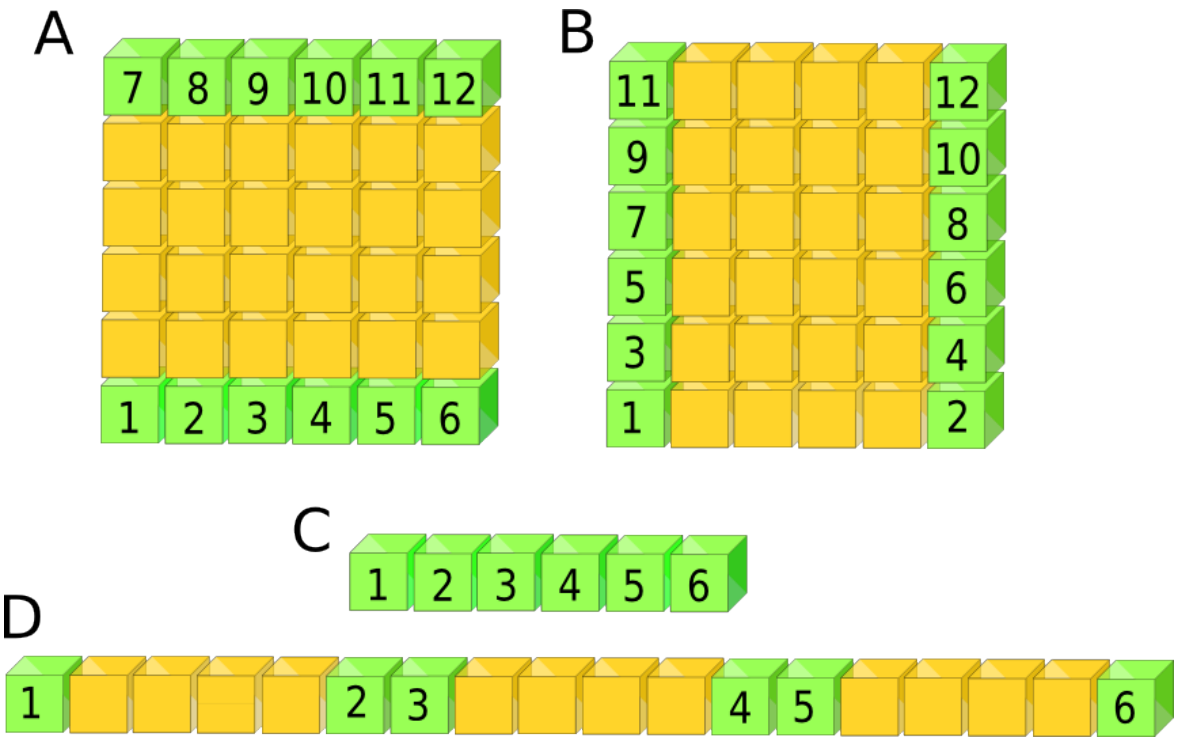


Figure 24: Access during halo exchanges: (A) contiguous top and bottom, (B) non-contiguous left and right, and first 6 bank accesses for top/bottom (C) and left/right (D).

The main difference between the two options is that the conditional statements require the kernel to be queued for all of $N \times M$ or $N \times M \times O$ cells, with some threads being ignored during execution. When calculating the location using offsets, it is possible to only

queue $(M-2a)(N-2a)$ or $(M-2a)(N-2a)(O-2a)$ cells (the yellow cells in Figure 23), where a is the depth of the halo cells. It was hypothesised reducing the number of threads requested may improve the performance by avoiding stalled compute units, but no performance gain was achieved. It appears from this result that the hardware is quite resilient to small numbers of inactive threads and either method will result in good performance.

Profiling showed a disparity in the performance of the halo exchanges performed during each iteration, for instance the top and bottom halo exchanges were, for some models, around 200% faster than the left and right. Figure 24 is represented in the row-major order used by the C programming language, and demonstrates the problem where memory access is contiguous in Figure 24-A, meaning that coalescence and proper cache utilisation are enabled by the algorithm accessing chunks of data like Figure 24-C. On the other hand, Figure 24-B is not contiguous, which leads to scattered reads in Figure 24-D that cannot be coalesced [5] and result in wasted caching.

In order to best take advantage of the memory layouts, it was essential to force neighbouring threads to perform contiguous access. This maximised the amount of cache utilisation, even for the scattered case where the potential was low.

9.2.1 *Conclusions*

The optimisation process found that, in all cases, carefully isolating data access and minimising it, as advised by Yan et al. [55], ensured the best possible performance for TeaLeaf. This of course strongly correlates with the understanding that it is a memory bandwidth bound problem, and reducing the amount of access directly improves the performance for all models.

9.3 MODEL-SPECIFIC

Using profiling data and recent research, it was possible to hone in on some interesting points of optimisation and significantly improve performance in some cases. The following discussion only includes the most interesting or novel of the optimisations performed.

9.3.1 *RAJA*

Once the initial port of RAJA was complete, the preliminary performance tests suggested that it was far slower, generally 500-600%, than the other CPU-targetting ports. This result was unexpected because a review of the code suggested that the port should essentially compile into a very similar binary representation as the OpenMP implementation.

Small rounds of testing were performed to check the overhead of using array indirection and/or lambda statements, but no discernable performance penalty could be determined. Core scaling tests (Figure 29 - *RAJA pre-fix*) showed that the RAJA implementation was not actually scaling with increased core counts, pointing to an issue with the framework's parallel implementation.

Profiling the application uncovered some interesting inconsistencies in the performance problems of the port. Notably, the CG solver has four main functions, three of which contain reductions and one that doesn't. The function that did not use a reduction was actually within 5% of the wallclock time compared to the OpenMP implementation, showing that it was running at least close to optimally. With this information it was possible to hone

in on the exact code that was causing the performance penalty in the RAJA framework, a representative sample of which is shown in Figure 25.

```
// Example of function exhibiting false sharing
void func(...)
{
    size_t num_threads = omp_get_max_threads();
    double sum[num_threads];

#pragma omp parallel for
    for(int ii = 0; ii < N; ++ii)
    {
        // Each thread accesses its own part of the array, but
        // the cache line has to be reloaded every time
        sum[omp_get_thread_num()] += 10;
    }
}
```

Figure 25: Example of false sharing in a function.

The RAJA custom reduction implementation included a similar case of false sharing. Allowing each thread to write to it's own private cell of a contiguous array meant that the reduction would return the correct result. However, this array is cached each time it is accessed, meaning each time a thread writes to the array all other threads incur a cache miss and have to reload the cache line. This actually results in a highly inefficient implementation but was resolved by writing a new implementation of the *forall_sum* function without the false sharing problem, then achieving greatly improved performance (Figure 29 - *RAJA*).

9.3.2 *OpenMP 4.0*

The OpenMP 4.0 port required a slow journey of optimisation, that suffered from a lack of supporting research. The problems that are presented in the available literature use compact examples that do not give a good indication of how to use OpenMP 4.0 in a larger application.

Initially, all loops within the C++ and Fortran code of the 3D application were embellished with OpenMP 4.0 target declarations, including directives to copy loop buffers to and from the device. This resulted in poor performance on the Intel Xeon Phi KNC (Table 4-1). In fact, the basic OpenMP implementation was compiled as a native application using the -mmic compiler flag and was able to perform the solve in 0.49s, 386x faster than the OpenMP 4.0 C++ implementation.

OPTIMISATION	OPENMP 4.0 (C++)	OPENMP 4.0 (FORTRAN)
1. First attempt	193.21s	236.11s
2. Data sharing	10.91s	49.73s
3. Single global target	2.74s	3.07s

Table 4: Results of Intel Xeon Phi KNC performance for CG solver on $20 \times 20 \times 20$ mesh.

The obvious culprit for such extraordinarily bad performance was data transfers, which often account for large slowdowns in memory-intensive applications [55]. To reduce the amount of data sent to and from the device, the **target data map** clause can persist data by declaring sharing within a structured lexical scope.

In order to perform optimisation for this problem, it was necessary to transform the main Fortran controlling code to perform only one read and write of the required buffers to the device for each step, persisting them over many function calls and iterations. The performance was significantly improved, achieving a 19x speedup (Table 4-2) for the C++ but much less for the Fortran implementation.

Even though the performance was greatly increased, there was still a large disparity between the performance of the native and offloading implementations. After significant investigation it transpired that each time a loop was entered that has been declared as a **target**, a considerable overhead was added just to invoke the loop. This turned out to be synchronisation performed at every target offload site, and reducing or removing it would improve performance [3], but required completely re-engineering the OpenMP 4.0 design.

Without any research to support this sort of change, a best guess optimisation was chosen, where all deep nested **target** invocations were removed and the entire Fortran controlling code was offloaded to the device. This meant declaring all Fortran and C++ functions as special offloadable functions, all global variables as target variables and again mapping all data for transfer at the beginning of the **target** invocation, the details of which are presented in section 7.6.

To reduce the amount of data transfer even further, data that did not need to be returned from the device was left in place, using the **map(to: ...)** directive, which doesn't read data back. The effect on the original code was significant but the results were much closer to the natively compiled target (Table 4-3).

Further testing demonstrated that the finely tuned OpenMP 4.0 implementation far outperformed the native implementations as the mesh size was increased up to the point of mesh convergence. Presumably for a $20 \times 20 \times 20$ mesh, the overheads of synchronisation and the original data transfer dominate the performance.

9.3.3 *OpenMP*

Upon execution, threads are mapped to cores, and this can be controlled using the environment variable KMP_AFFINITY, which has a number of parameters that can co-ordinate the binding explicitly [41]. By testing the OpenMP implementation over 50 test runs per parameter choice, it was possible to determine the effect of altering this setting on the performance of the application (Table 5).

KMP_AFFINITY SETTING	MIN. RUNTIME (S)	AVG. RUNTIME (S)	VARIANCE
none	8.452	9.739	3.463
compact	8.179	8.335	0.008
scatter	8.551	9.273	0.322

Table 5: Effect of altering the KMP_AFFINITY parameter on OpenMP performance.

The 'scatter' setting does improve the variance of the runtimes, but the minimum performance is slightly worse than choosing 'none'. The 'compact' setting, however, improves

the performance by around 12%, but the minimum runtime is only improved by 3%, and the greatest effect is on the variance of the runtimes over 50 samples. Importantly, when choosing 'none' the highest observed runtime was 19.2 seconds, but the 'compact' setting only took 8.6 seconds in the worst case. This consistency is important when scaling applications up to larger mesh sizes, where the difference can be significant enough to cause a noticeable instability in performance and make timing inaccurate.

The KMP_AFFINITY environment variable can also be passed a 'verbose' flag, which prints out the binding performed at runtime. This showed that the 'compact' setting binds cores in total order (thread 0 to core 0, thread 1 to core 1, etc.) and 'scatter' appears to disperse neighbouring threads evenly distanced from each other (thread 0 to core 0, thread 1 to core 8, thread 2 to core 1, thread 3 to core 9 etc.).

When the verbose flag is provided with the setting 'none', the output suggests each thread is offered the entire pool of cores. Presumably this means that the threads bind to cores non-deterministically, perhaps on a first come first served basis, which makes sense given the high variation but equivalent minimum runtime. Ultimately, the best setting for this application is 'compact', and it is believed that this is caused by an improved utilisation of cache that comes from creating two groups of neighbouring threads each residing in a separate non-uniform memory access (NUMA) region.

9.3.4 *Conclusions*

Optimising each of the different ports required differing levels of individual attention, and in particular OpenMP 4.0 and RAJA required significant work to achieve optimality, and Kokkos required very little additional work. Overall, the optimisation process has been successful where unanticipated results have been greatly minimised, and good performance has been achieved by the ports on most devices.

9.4 UN-OPTIMISABLE DISCOVERIES

While attempting to optimise the ports, there were some strange results that could not be resolved and appeared to be caused by a bug or implementation problem. In all cases the tests were repeated many times (at least 10), and at different times of the day to ensure that the results were not caused by environmental factors.

9.4.1 *Fortran PPCG*

The Fortran implementation is the original OpenMP port of TeaLeaf, and generally exhibits very good performance, except for a strange bug with PPCG solver when compiled with the GNU compiler suite.

Figure 26 clearly shows this issue, noting that the same graph plotted for other models demonstrates an even split between the models, as seen in the 32×32 case. This performance profile is unexpected, and does not replicate when the exact same code with the exact same configuration is compiled with the Intel compiler suite. It is expected that, through targetted profiling and debugging, it may be possible to find the cause of this bug, however it was not possible during this research project and will be proposed for future work.

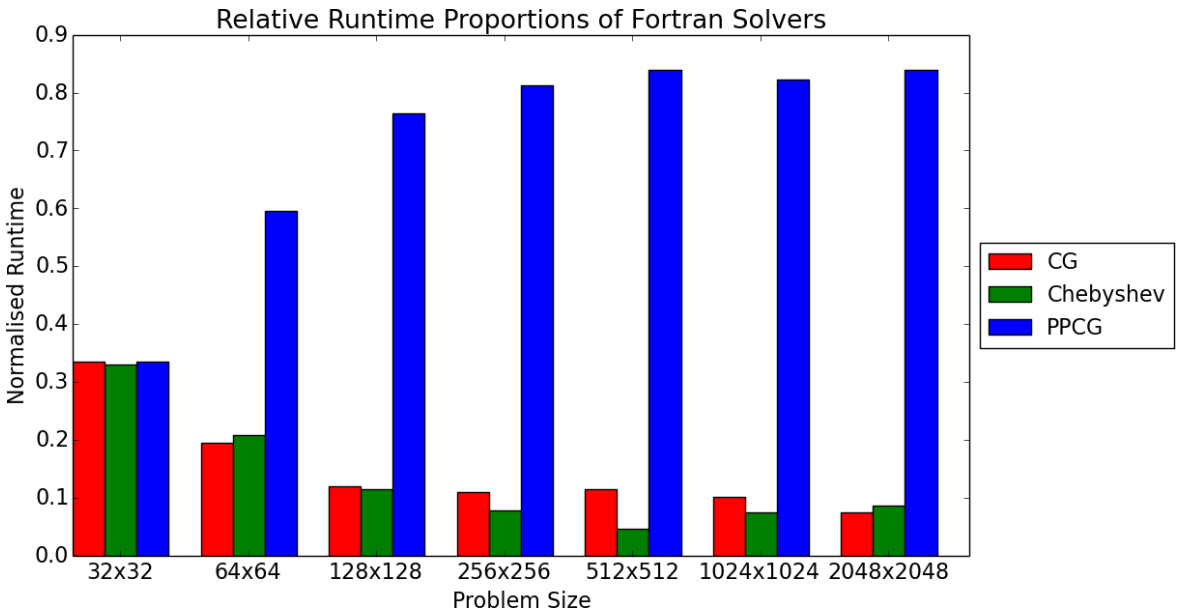


Figure 26: The one node runtime with clear patterns of divergence.

9.4.2 *OpenCL CPU Outliers*

A regular problem encountered during the data collection phase was outliers, but by taking the minimum encountered absolute runtime value, it was possible to avoid the risk of inflated runtime results affecting the observed result.

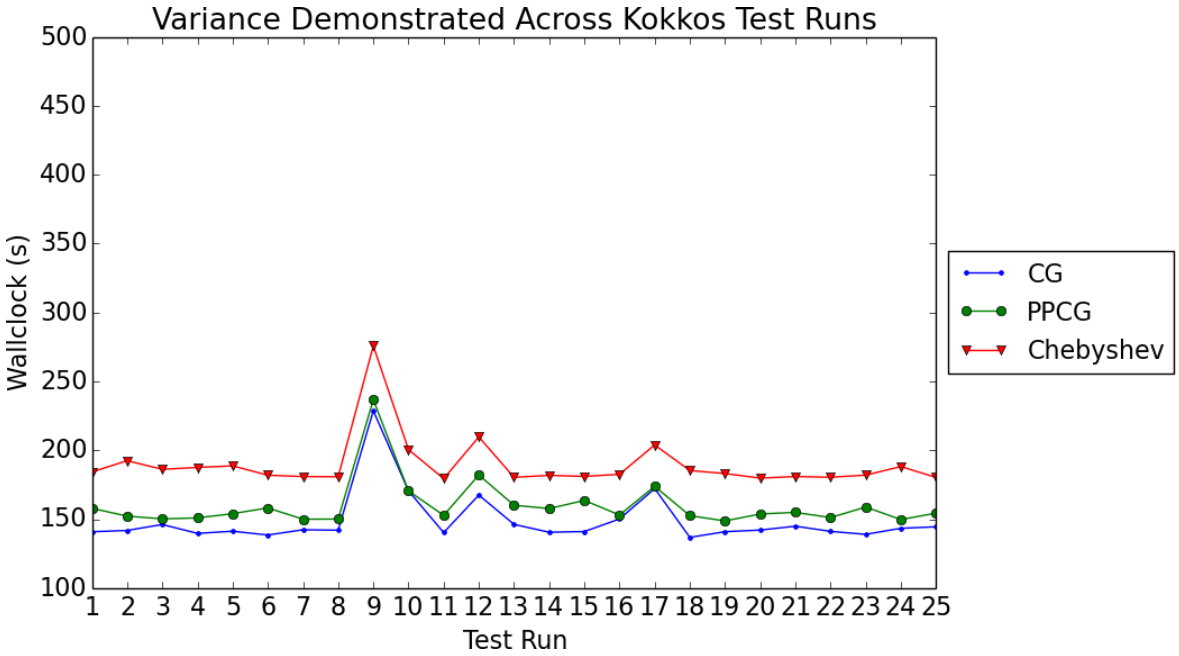


Figure 27: Demonstrating the low variance of the Kokkos CPU runtime including outliers.

This of course relies upon the intuitive assumption that there exists a lower bound on this potential runtime, which represents the best possible performance on a particular architecture. It was anticipated that the majority of runs should fall within a small distance

from this lower bound, given that the application is being run on carefully tuned nodes of a supercomputer, where environmental factors are heavily reduced.

Figure 27 demonstrates the runtimes observed over 25 test runs, where the version and configuration was kept consistent for the whole test. The results clearly show that the variation is very low, except for the occasional spike, and the results are very close to the mean.

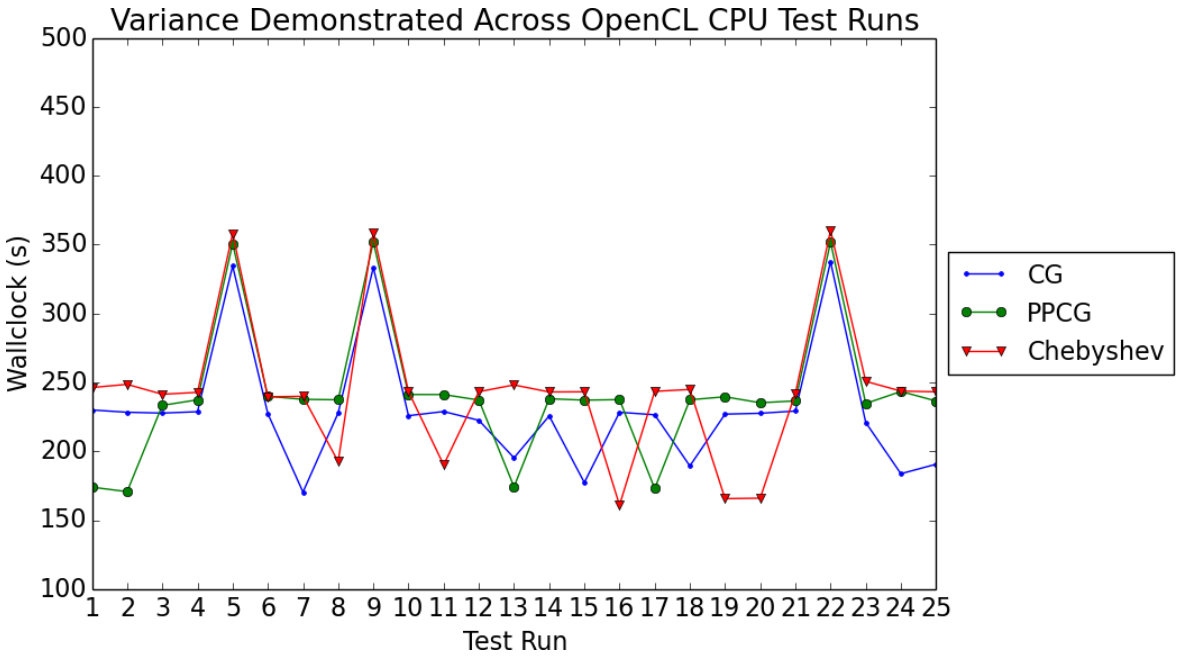


Figure 28: Demonstrating the high variance in the OpenCL CPU runtime including outliers.

Figure 28 on the other hand, is the same test run for OpenCL on the CPU, and it is clear that there is a serious problem affecting the performance. It is a highly unanticipated result that some of the results are twice as fast as the majority. An important observation is that the Kokkos implementation has correlated spikes, where all three solvers experience the same performance problem, suggesting something is happening at the node level. This can be seen three times in OpenCL's case, but the spikes that fall below the mean line are not correlated, representing an issue with the individual implementation.

As the solutions generated were correct and identical each time, it can only be assumed that there is either a non-deterministic issue with the application's OpenCL implementation, or an external factor is affecting the application. Given that there has to be a lower bound on the potential runtime, at most 14 of the 75 test runs actually represent the true peak performance of the application.

Interestingly, a known difference between those implementations is that the CPU version is built upon Intel Thread Building Blocks, which uses a non-deterministic "work-stealing-based task scheduler" [27], and investigating this issue is suggested as future work.

9.4.3 *Conclusions*

Although these issues could not be resolved as part of this research project, they represent some interesting avenues for future research. In the long term, it is important that the problem facing the OpenCL CPU implementation is discovered as this has the potential to affect production codes if it is not simply an isolated bug in TeaLeaf.

10 RESULTS

In most cases the tests have been repeated for the 2D and 3D ports across all solvers and large ranges of mesh sizes, meaning that there is far too much information to present here, and so only highlights are included, and some limited additional results are presented in Appendix A. Also, the project deals with a significant number of different ports at once and so visualisation through graphs has been preferred over tabular presentation.

It must be re-affirmed that the peculiarities in the data are not caused by lack of repetition as each test has been run at least five times to achieve the final result. Throughout this section a discussion is presented about the potential causes for interesting aspects of the data, however in some cases it is not possible to give definitive reasons. This may be because there is some transient effect or that there are simply too many factors affecting a result and not enough time or information to determine exactly which is the cause.

10.1 CORE SCALING TESTS

In order to observe the ability for the CPU models to scale when provided with additional resources, a test was run to calculate the performance as the number of available cores increased from 1 to 16 on a pair of Intel Xeon E5-2670 CPUs (Figure 29). A mesh size of 512×512 has been chosen because, as shown later in section 10.2, it is an optimal data load for all of the CPU implementations, where the whole mesh fits into fast cache. Please note that OpenCL CPU could not be included because the number of threads cannot be changed through the OpenCL APIs [19].

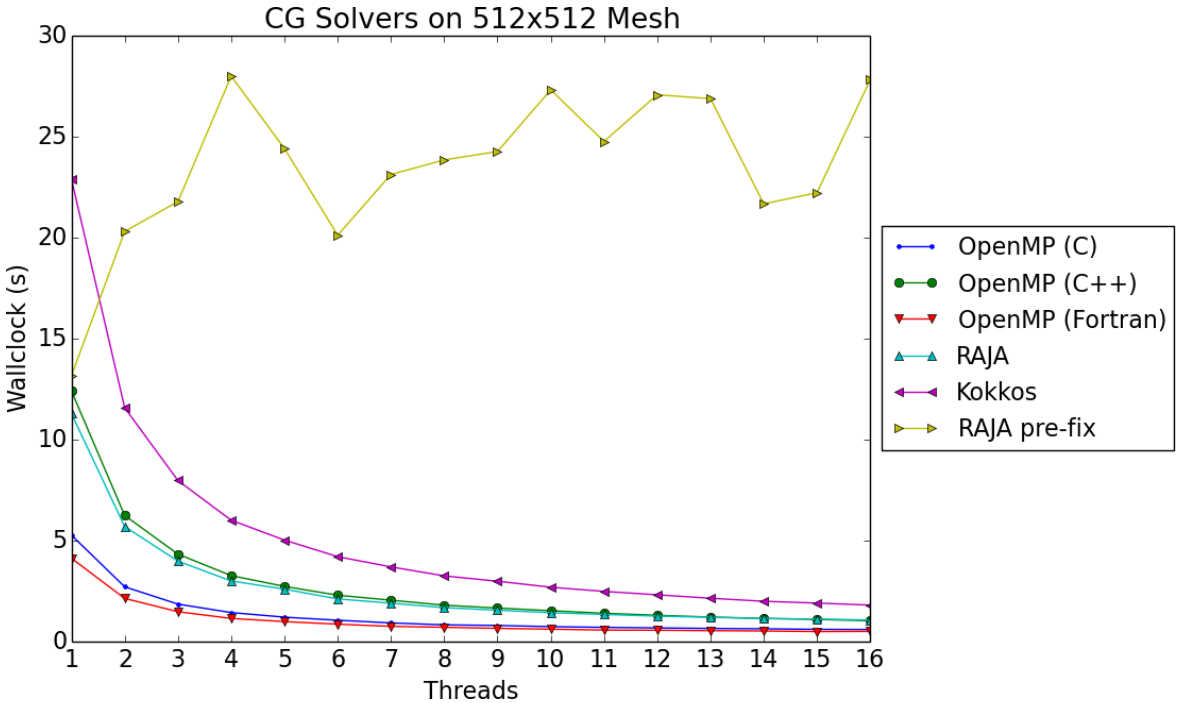


Figure 29: Core scaling tests demonstrating that RAJA does not scale correctly.

As discussed in subsection 9.3.1, the RAJA pre-fix result in Figure 29 demonstrates the impact of false sharing upon the original implementation, and the fixed version exhibits the same growth as the other OpenMP implementations.

Interestingly, the OpenMP C++ and RAJA ports achieve identical growth and performance, which is indicative of the similarity of their executable code. While RAJA essentially compiles into OpenMP, it does use modern C++ features that could carry performance overheads, but the data suggests that they are insignificantly small. Kokkos scales similarly to the other implementations, however the performance is considerably worse at this mesh size, which is further investigated in section 10.6.

The result shows that the wallclock performance improves roughly logarithmically when given additional thread resource. This result does at first appear slightly counter-intuitive, as you would likely expect that given additional resources the performance should scale linearly for a well optimised application such as TeaLeaf. Of course, this growth is based on adding an additional thread, where the work is then distributed between all threads evenly, only allowing a reduction in runtime relative to the portion of the work split.

This effect can be better understood by observing the same data but plotting speedup instead of absolute runtime.

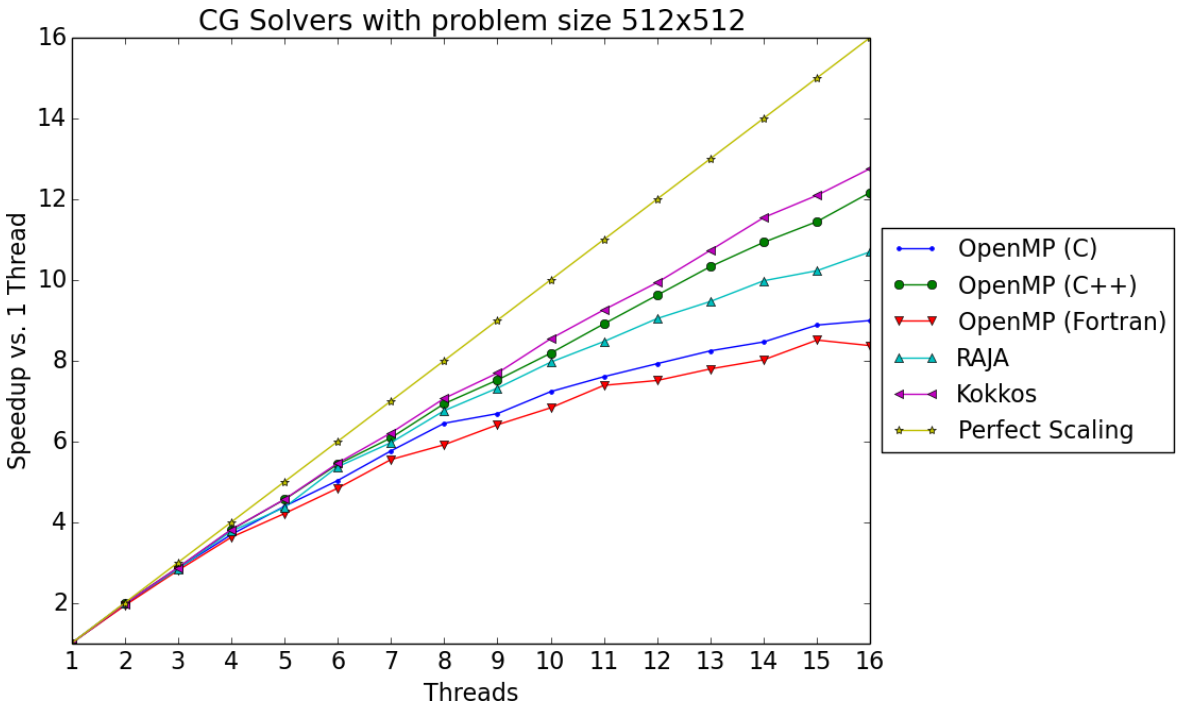


Figure 30: Speedup as threads are increased for OpenMP driven implementations.

The results in Figure 30 clearly show that each time a thread is added, the speedup increases, even though this was not immediately apparent when observing the runtime. The perfect example of scaling is the main diagonal, where the resulting runtime is 16 times faster after being split across 16 cores. It can be seen that the TeaLeaf application scales well for all models when provided with up to 4 threads. However, this scaling diminishes and only ever reaches between 8× and 13× speedup by the time that all 16 threads have been engaged.

An important observation is that Kokkos achieves the best speedup even though it was the worst performing port. There are several characteristics of an application that may affect scalability, but many of the potential contributors to the strange performance are

made irrelevant by the fact that Kokkos and the other implementations are configured identically and have conserved core logic.

Amdahl's law [1], for instance, states that the maximum possible parallel speedup is limited by the fraction of an algorithm that is purely serialised. In TeaLeaf's case, the application has been stringently developed to remove as much serial execution as possible from the algorithm, allowing more than 99% of the computation to run in parallel. The parallel programming models such as Kokkos and RAJA may actually increase the amount of serialised code, reducing the scalability, but they cannot possibly reduce the serial portion relative to the OpenMP implementations.

Other potential causes of this disparity in speedup are the utilisation of memory, where some models better utilise the CPU L1-3 caches, or the effect of overheads being hidden through parallelisation. However, it can be seen in the visualisations that good scaling is not necessarily reflective of good performance. By considering both visualisations at once, it is clear that the consistency of the scaling is more important than the total order between models. Later tests make thorough comparisons between the models based on runtime performance when they are provided with all 16 threads.

10.1.1 *Conclusions*

This test allowed an immediate discovery of a fault with the original RAJA implementation which, once fixed, allowed RAJA to scale identically to the other OpenMP-based implementations. Kokkos also scales identically, even though the performance is worse, slightly closing the performance gap by 16 threads. Importantly, all of the models are scaling in a consistent manner, which enables a successful continuation of the testing.

10.2 EVEN STEP

The "even step" testing was performed on the Blue Crystal supercomputer using the CPU, GPU and accelerator (PHI) listed in section 8.1. The mesh size was incremeted in steps of approximately 100,000 cells (as close as possible with a square mesh), providing a representation of the scaling of each model on a single node. Please note that some of the Intel Xeon Phi KNC results are not included in full in the graph because they are so much slower than the other results that it would obscure important details.

Originally this investigation had only evaluated up to 800,000 cells and it gave very little information, but it was later discovered from the 'power of 2' test (section 10.3) that some models dramatically changed in performance at approximately 1,000,000 cells, and extending the test uncovered far more interesting patterns.

It is clear from Figure 31 that for 1,000,000 cells and under, the OpenMP Fortran and C ports dominate, demonstrating far better performance than the other devices. This effect can be explained by the fact the those ports are utilising the available CPU cache to the greatest extent possible, which has lower latency and higher bandwidth than the memory available to the other devices. RAJA and OpenMP C++ demonstrate slightly worse performance than Fortran and C, showing that the C++ ports have generally reduced performance (section 10.5).

The CPU models (see legend) all begin to rise at around 800,000 to 1,000,000 cells. This is caused when the mesh size is so large that it cannot fit into the high bandwidth CPU L1-3 caches, meaning that the models that use a CPU have to spread some of the data into

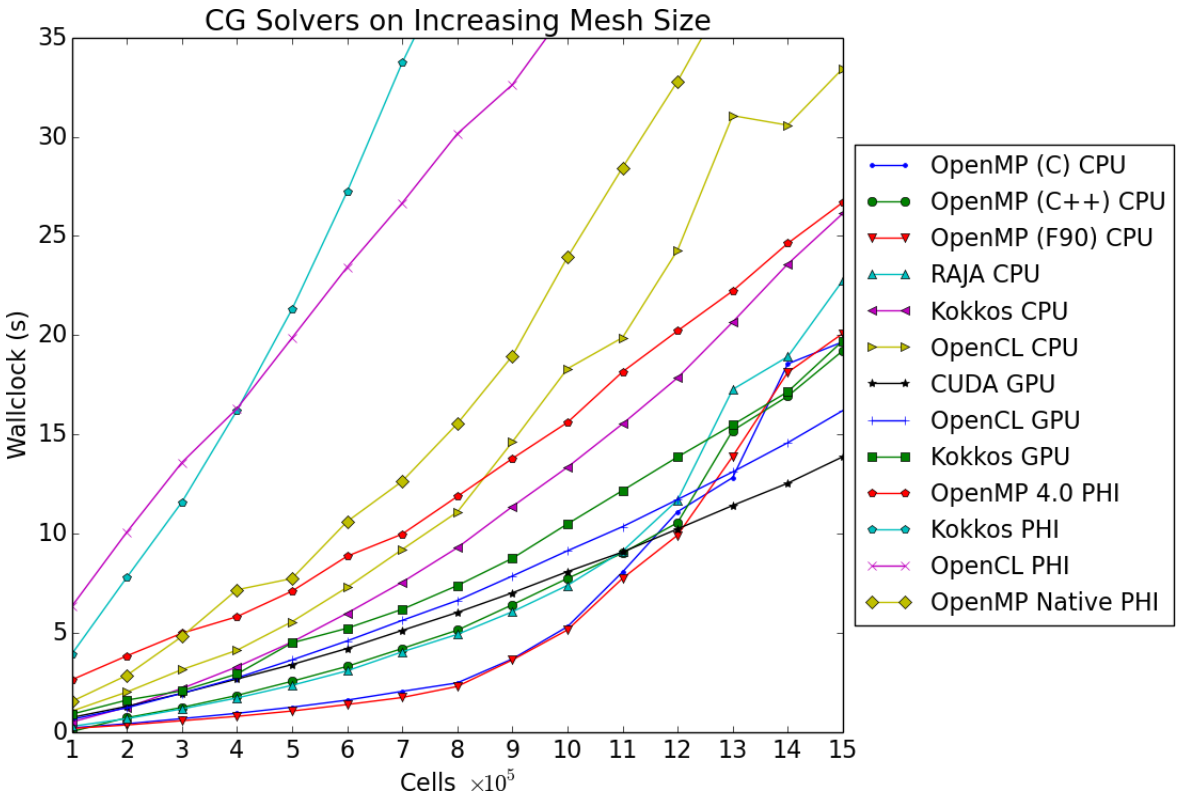


Figure 31: Even step scaling test performed for CG solver with 2D ports.

DRAM. Because TeaLeaf is memory bandwidth bound and the bandwidth is much lower to external memory (76.2 GB/s for the pair of CPUs used in this test), the application will be bottlenecked by the data transfer, exacerbated by the increasing mesh size.

The GPU models grow at a much shallower and nearly linear rate, while the OpenMP 4.0 port is slower and slightly more irregular but does demonstrate roughly linear growth. This result is very important as it demonstrates the ability for the higher bandwidth devices to handle increasing mesh sizes, as discussed in chapter 2.

The OpenCL CPU implementation appears to grow quite erratically and eventually achieves much worse performance than the other CPU-targetting implementations. An important aspect of this test is that it requires the applications to work with unusual mesh sizes, even having side lengths that are prime numbers. It was considered that the erratic performance displayed by the OpenCL implementation may have been an issue with the decomposition of the problem into work groups, however, the GPU implementation of OpenCL uses identical code and does not suffer from the same problem. This chaotic trajectory is likely caused by the OpenCL variability discussed in subsection 9.4.2, but the poor performance seems to be specific to the CG solver, as later testing shows that the other two solvers demonstrate better performance.

While the results for the Kokkos, OpenCL and OpenMP Intel Xeon Phi KNC targetting ports aren't fully displayed because of their vastly inflated runtimes, it is clear that their trajectories are very sharp, and demonstrate far worse performance than the other models. The OpenCL PHI implementation grows roughly linearly over time, but the Kokkos and OpenMP ports begin to curve, demonstrating worsening performance as the mesh size increases.

These same results are observed across all three solvers and the 3D implementations. In some cases the growths are more chaotic but in general the order and change points remain the same.

10.2.1 *Conclusions*

This test has demonstrated that most of the models are able to grow efficiently given increasing mesh sizes, but that some of the implementations are more stable than others. In particular, the CPU implementations grow linearly until the threshold of roughly $1,000,000$ cells is crossed, whereas the GPU implementations demonstrate almost perfect linear scaling throughout, and dominate the performance at larger mesh sizes.

Similarly, the OpenMP 4.0 and OpenCL PHI implementations display linear growth representative of a high memory bandwidth device, but the Kokkos and OpenMP MIC native implementation experience the sharpest growth and demonstrate the worst performance in the long term.

10.3 POWER OF 2

This test continues on from the even step test to provide a bigger picture of the scaling of each model as the mesh size increases. The results have been collected up to the point of mesh convergence, 4096×4096.

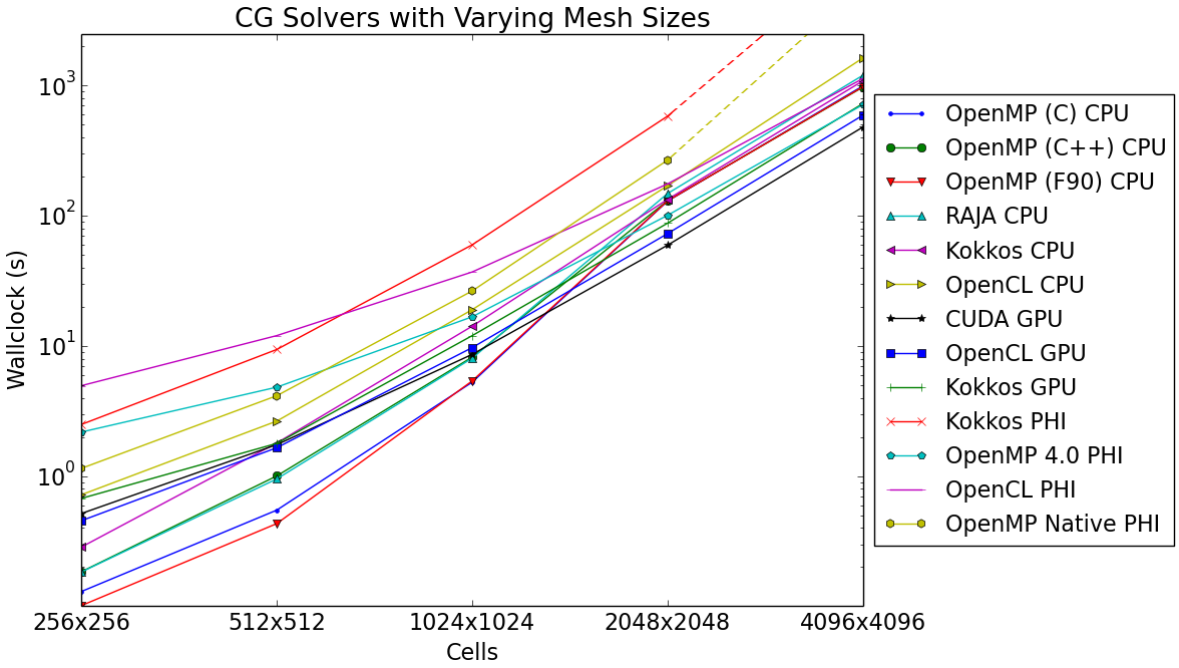


Figure 32: Scaling the mesh size in powers of 2 for CG solver.

The result for OpenCL on the CPU is significantly worse than the other implementations, and it must be noted that this performance problem is worst for the CG solver, but no cause for this performance disparity has been uncovered. The other CPU models demonstrate a consistent growth pattern, clustering between 2048×2048 and 4096×4096. Prior to that point, the C and Fortran OpenMP implementations are far superior to the other options, with RAJA and OpenMP C++ showing slightly worse performance. Inter-

estingly, the Kokkos implementation is worse than the GPU-targetting implementations for the 512×512 mesh size, while the other CPU implementations are not overtaken until 2048×2048. Further to this, the Kokkos implementation actually grows linearly as the mesh size is increased, perhaps indicating that the implementation is not effectively utilising fast CPU cache for the smaller mesh sizes.

The GPU ports and OpenMP 4.0 all deliver reasonably poor performance compared to the OpenMP implementations for smaller mesh sizes (less than 1,000,000 cells), but they become the best options at the point of mesh convergence. The OpenCL PHI implementation actually suffers from poor performance right up until mesh convergence, at which point it achieves performance equivalent to the CPU ports. This could demonstrate that there are overheads dominating the performance for small mesh sizes, but it is surprising that the port did not beat the CPU implementations on the 4096×4096 mesh.

Unfortunately, the Kokkos and OpenMP native dashed results are only estimated from a small number of iterations. As previously stated, there is a limit of 1 hours runtime when testing on the Intel Xeon Phi KNC, and the CG solver for the 4096×4096 requires several hours to complete, making a full run impossible. The results clearly show that the natively compiled options are deficient for all problem sizes when compared with the other models but that Kokkos demonstrates a larger overhead than OpenMP native.

10.3.1 *Conclusions*

This test extended the evidence showing that the CPU versions perform very well at small mesh sizes, but as the mesh is increased, the GPU and accelerator ports are able to handle the data more efficiently. As mesh convergence is the point where the real science is fulfilled, these results clearly advocate the use of high bandwidth devices for such memory bandwidth bound problems, and OpenMP 4.0, CUDA, OpenCL GPU, and Kokkos GPU can all support this very well.

10.4 COMPARISON BY COMPILER

The GNU and Intel compilers were compared to investigate if there was any significant difference in performance resulting from the compilation of the models. This test is only relevant for a subset of the ports: OpenMP, RAJA and Kokkos, which are directly affected by the choice of compiler. The other models are either tied to a particular compiler or have unadjustable mechanisms, for instance, the GPU ports all utilise the NVIDIA *nvcc* compiler.

Figure 33 shows that the Intel compilers outperform the GNU compilers for every single model. This does make sense because the tests are being performed on Intel CPUs and internally use Intel's OpenMP, and the compiler has been specifically tuned to optimise applications for Intel technologies [44]. For this mesh size, the difference is around 10% on average which is a considerable performance improvement for larger applications of the algorithms.

To further validate the results, the same experiment was performed for different mesh sizes as well, and the results for the 4096×4096 mesh size are presented in Figure 34. As the mesh size is increased, the Intel compiler produces even more efficient code, even exceeding a 33% improvement for OpenMP Fortran Chebyshev. This level of performance improvement is surprising and suggests that the Intel compiler is aggressively optimising memory access and generating well vectorised code.

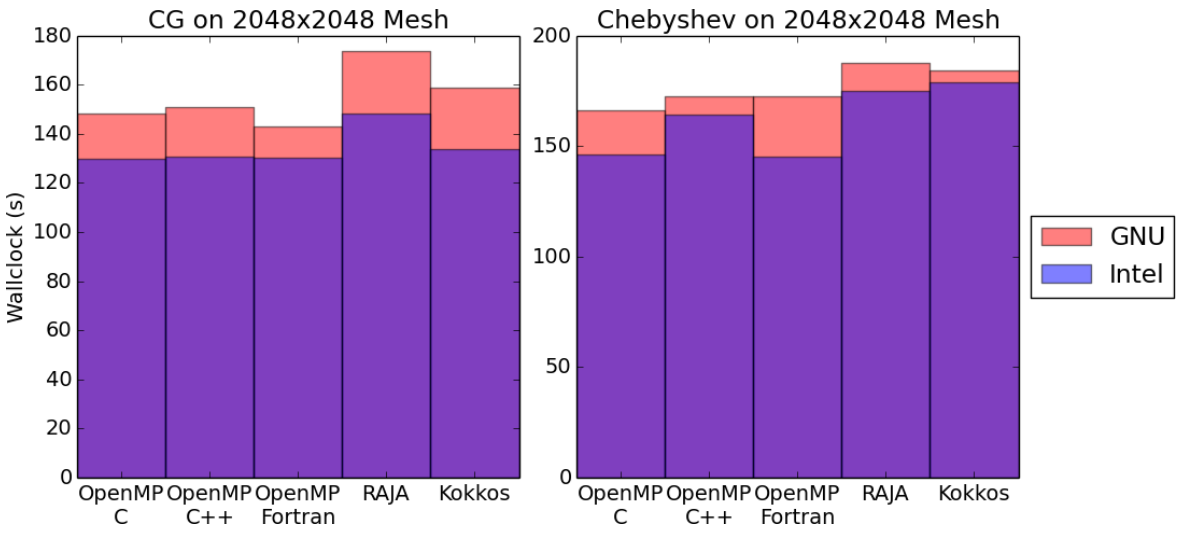


Figure 33: Comparison of the runtime for solvers with a 2048×2048 mesh.

RAJA suffers from worse performance with the Intel compiler at mesh convergence for the Chebyshev solver, achieving identical performance for both compilers. It would appear that the way that RAJA is being compiled for the Chebyshev solver being optimised as well by the compiler. This might be caused by deficiencies in the compiler's handling of lambda statements, which was outlined by Hornung et al. [23] as a potential risk to RAJA.

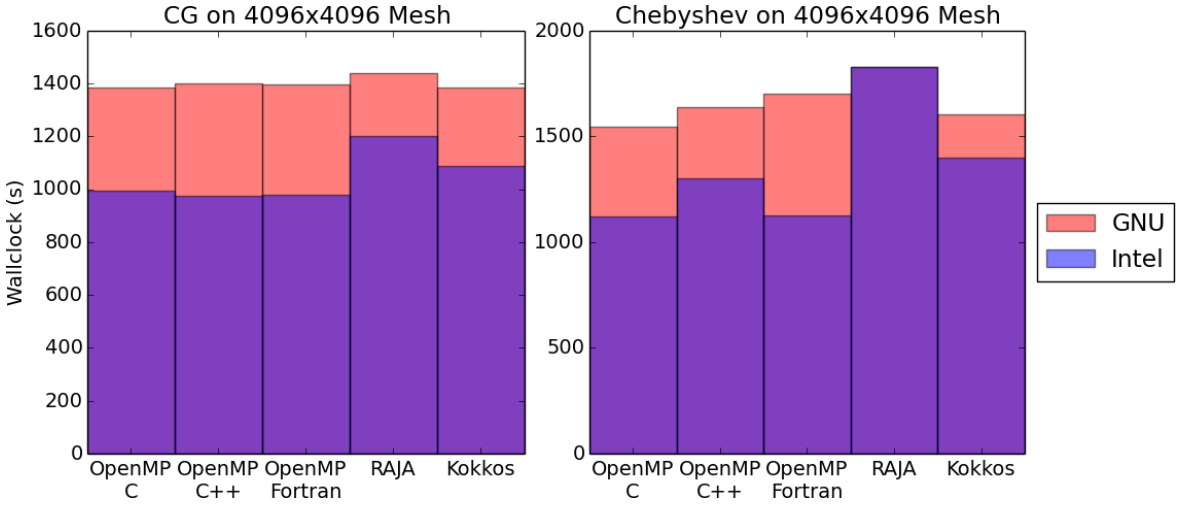


Figure 34: Comparison of the runtime of CG and PPCG solvers on the GNU and Intel compilers.

10.4.1 *Conclusions*

For TeaLeaf, the Intel compilers are more successful at compiling the tested range of ports efficiently in all cases for all solvers. Based on this information, it gives strong evidence to suggest that the Intel compilers are a better option than the GNU compilers when using OpenMP and Intel architecture, of course the same fluctuations may not apply to devices from other vendors. Although the exact cause of the RAJA Chebyshev performance penalty could not be isolated during this research project, it would be important future work to uncover which aspect of the Chebyshev solver is leading to this reduction in performance.

10.5 FORTRAN VS. C VS. C++

It was discovered that the C++ implementation was up to two times slower than the Fortran version for some mesh sizes. Devolving the OpenMP C++ implementation into a C application lead to an impressive performance improvement, for instance the CG solver on a 512×512 mesh improved from 0.99s to 0.56s. This OpenMP implementation matched the performance of the Fortran implementation, but it was not known what caused the disparity in the first place.

Figure 35 demonstrates that the C++ implementation suffers from an unanticipated performance hump between 128×128 and 1024×1024, but that this difference completely disappears from this point. Interestingly, this point of change between 1024×1024 and 2048×2048 recurs throughout the results discovered for TeaLeaf.

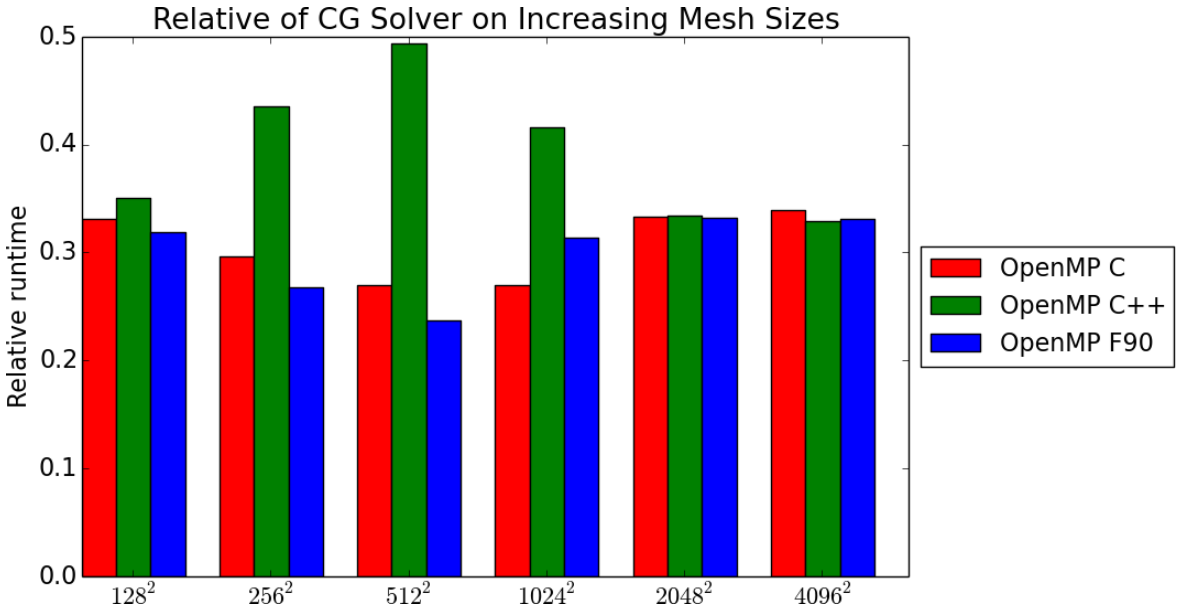


Figure 35: Performance hump discovered for C++ implementation of TeaLeaf.

The C++ implementation had a design incorporating a set of C interface functions that were called by the Fortran controlling code, but those interface functions called member functions of a global C++ class. The effects were particularly noticable on smaller mesh sizes and there were two suspected culprits: (1) the overhead of maintaining and calling class member functions underpinned by C++ v-tables, and (2) the fact that the **restrict** keyword could be added to the C implementation, which avoids the inefficiencies of alias checking [31].

This was tested by first removing calls to member functions, which had no impact on the performance, and then each function call was altered to accept raw pointers embellished with the **restrict** keyword, which also didn't affect the performance.

10.5.1 *Conclusions*

In the end, this result demonstrated that the performance problem would be encountered when the C application was simply compiled as C++. This leaves some uncertainty, but it is hypothesised that the disparity is caused by deficient or excluded optimisations of the C++ compiler, and proposed as a topic for future investigation.

10.6 RUNTIMES BY DEVICES

This section presents the performance achieved by the ports on the three distinct devices tested throughout this project (section 8.1). Comparing the results of each model on its supported devices exposes the performance penalties associated with the performance portable implementations. For the industrial partner, AWE, this particular analysis provides a prediction of the performance that can be achieved if their real scientific codes were re-designed, at least in part, to use one of the modern parallel programming models.

10.6.1 *CPU Ports*

For the CPU implementations it is essential that both the 512×512 and 4096×4096 mesh sizes are presented because they show very different results.

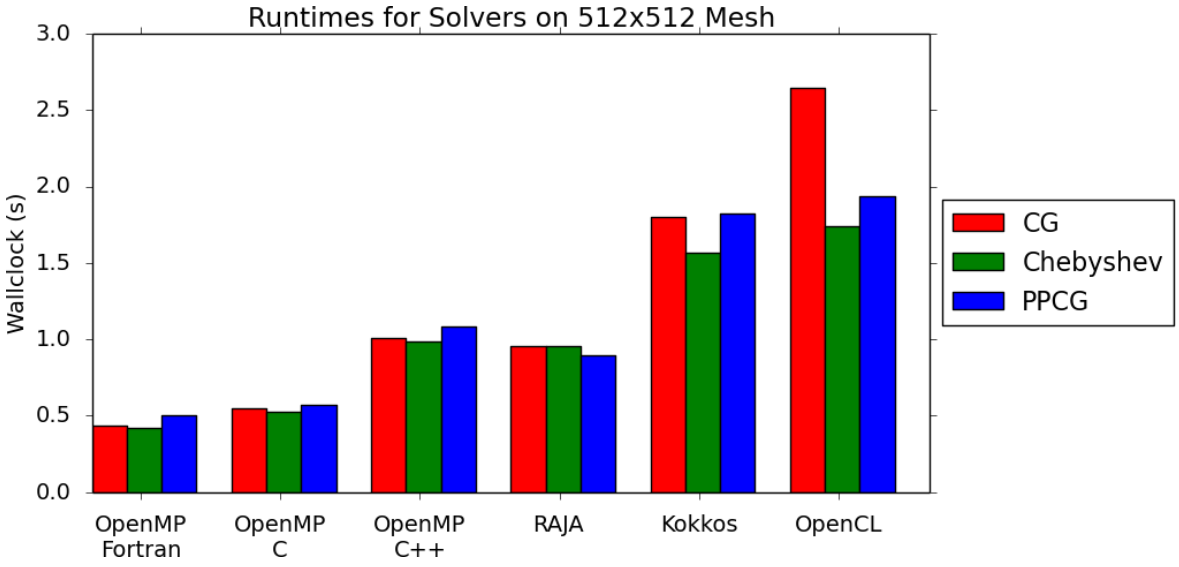


Figure 36: Results of applicable models on 2 Intel Xeon E5-2670 CPUs.

Figure 36 demonstrates that there is a significant gap between the best performing Fortran implementation and the worst performing OpenCL implementation. It can be noted that RAJA is the fastest performance portable implementation, slightly beating the OpenMP C++ implementation. It is hypothesised that this small performance difference is caused by the fact that the C++ implementation only parallelises over the outer loop of each function, whereas RAJA can parallelise over the entire loop by using array indirection, allowing a more optimal distribution of threads for the smaller mesh size. Both the OpenCL and Kokkos implementations achieve similar levels of performance, which is around 3 times slower than the Fortran implementation.

In isolation these results would be fairly damning of Kokkos and OpenCL, which may be poorly utilising the fast CPU cache or even introducing runtime overheads, but are on average 3 times slower than the device-tuned implementations. Figure 37 shows the same models at mesh convergence and offers a totally different perspective where, excepting some outlying results, all of the performance portable implementations are able to achieve performance to within 20% of the device-tuned OpenMP implementations.

In contrast to the results at 512×512 mesh size, the three performance portable models demonstrate very good performance, where the Kokkos implementation is the fastest

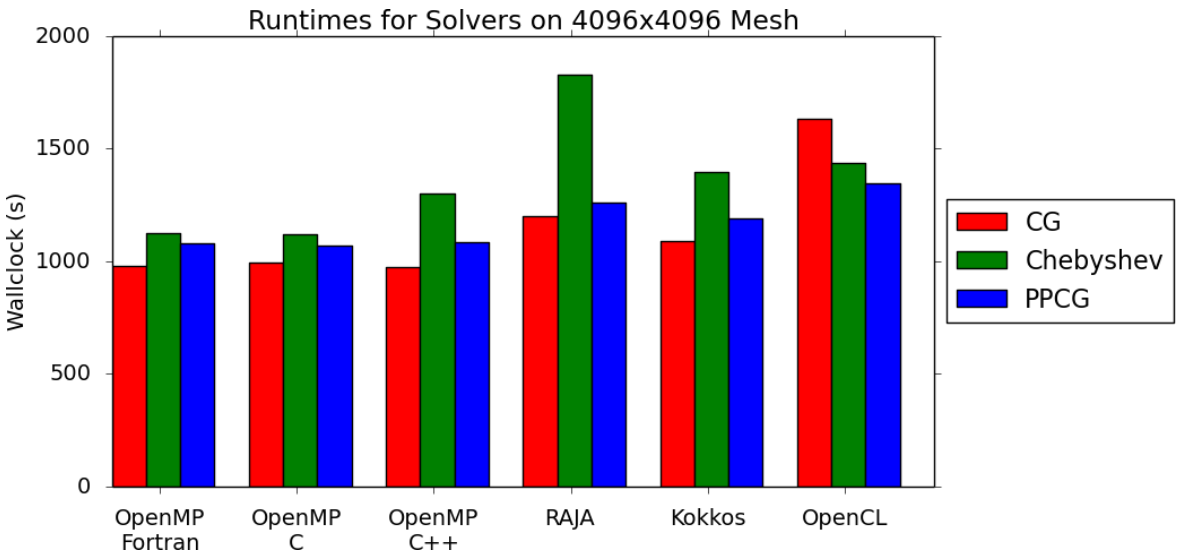


Figure 37: Results of applicable models on 2 Intel Xeon E5-2670 CPUs.

and most consistent. The difference between all of the models is greatly reduced, and it is thought that this is most likely because the CPU cache is no longer influencing the performance because of the volume of data being operated on. As this point is reached, those models that were experiencing improved performance from cache optimisation can no longer benefit and the performance somewhat standardises.

All of the solvers exhibit worse relative performance for Chebyshev solver and whatever is causing this problem is particularly affecting the RAJA port, for all mesh sizes from 1024×1024 onwards (Appendix A). Further to this, the OpenCL port performs poorly for the CG solver in comparison to the other implementations, which could be caused by the timing issue discussed in subsection 9.4.2. It is expected that with focussed profiling and investigation it should be possible to determine the underlying causes of the performance problems, which is proposed as future work.

10.6.2 *GPU Ports*

Overall, the results in Figure 38 are quite compelling, showing that at mesh convergence the performance portable models are able to compete well with the device optimised CUDA implementation. Apart from a bug with the Kokkos handling of the CG solver, the performance is almost identical across the 3 models, and additional results shown in Appendix A further demonstrate that the GPU implementations provide consistent performance regardless of mesh size.

The poor Kokkos CG result is seen across all mesh sizes, and has been profiled to attempt to determine the underlying cause of the performance issue. This uncovered a problem with the halo exchanges which take 2s on average for this mesh size on the GPU, but around 100s for Kokkos CG. As the code and configuration are consistent when targetting the CPU, which exhibits very good CG performance, it may well reflect an issue with the Kokkos CUDA implementation or the techniques used in the port.

It is important to acknowledge that the presented results focus only on NVIDIA GPUs, excluding AMD GPUs, which OpenCL can capably target [33]. As and when Kokkos and

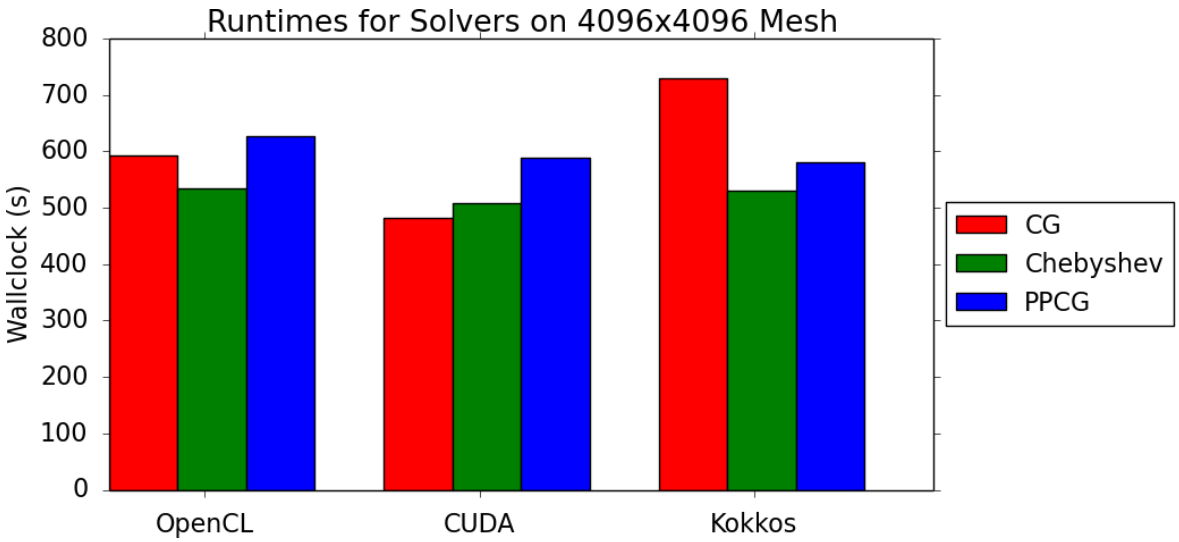


Figure 38: Results of applicable models on the NVIDIA K20 GPU.

RAJA are extended to execute on new devices, it will be necessary to continue this research to further evaluaate their ability to perform across heterogeneous devices.

10.6.3 *Intel Xeon Phi KNC Results*

The final device is the Intel Xeon Phi KNC, but the results could only be shown for the 2048×2048 mesh size because of a limit on the maximum allowable runtime when testing on this particular device.

In the current literature there is a lot of discussion about the relatively poor performance exhibited by the Intel Xeon Phi KNC [12, 34]. Interestingly, several of the models were able to target this device, which is fairly unique architecturally, and the results demonstrate a significant disparity. Please note that this is the only device that the OpenMP 4.0 standard has official support for targetting, which leaves the model in a position of relative immaturity similar to RAJA.

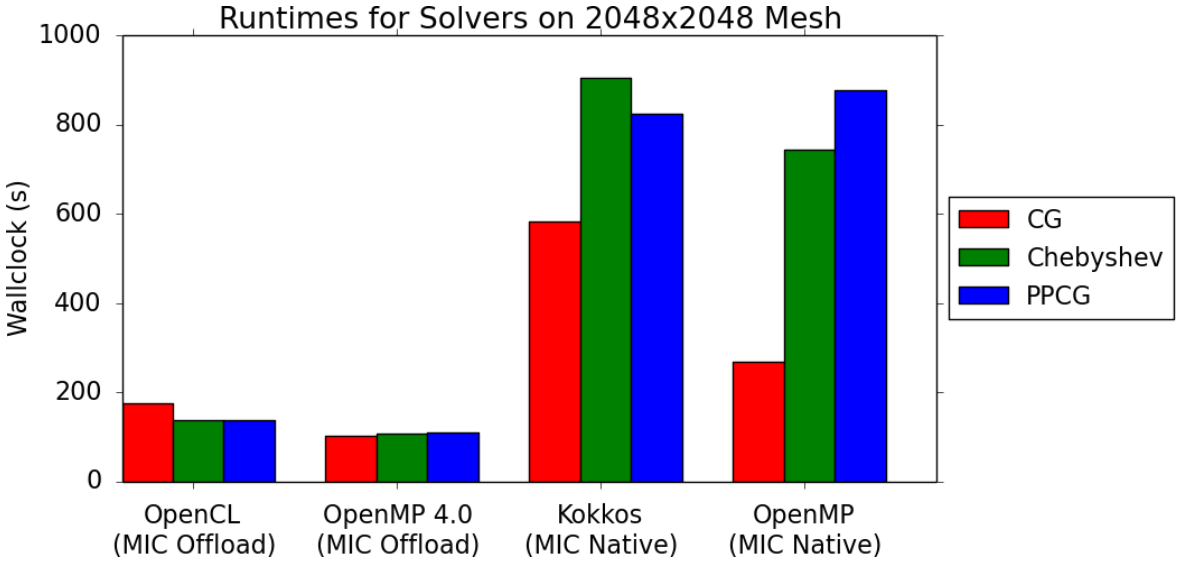


Figure 39: Results of applicable models on the Intel Xeon Phi KNC accelerator.

Both the OpenMP 4.0 and OpenCL models utilise an offloading strategy where code segments and data are sent to the accelerator device at runtime to handle the processing. The OpenMP and Kokkos MIC native implementations both compile the application to be run in its entirety on the accelerator device. Figure 39 shows that the performance difference is significant, where the natively compiled models perform particularly poorly, but the OpenCL and OpenMP 4.0 implementations actually perform very well.

It can also be noted that both OpenMP 4.0 and OpenCL continued to achieve consistent performance at mesh convergence, and OpenMP 4.0 was faster than on a CPU but slightly slower than the GPU implementations, which is a good result. The additional data in Appendix A show that OpenCL does achieve very close performance to OpenMP 4.0 throughout, but suffers from an unknown issue with the CG solver.

10.6.4 *Conclusions*

Overall, the performance between the models was fairly consistent, with both the CPU and NVIDIA GPU implementations generally falling within a 20% performance band. The models that were natively compiled for the Intel Xeon Phi KNC devices did present a significant performance penalty and so would not be recommended for production use. The results have clearly demonstrated the power of the performance portable implementations whilst highlighting some important issues with their implementation.

As previously mentioned, the KNC architecture will be replaced in the long term by the Knights Landing architecture, which makes the current KNC many-core approach somewhat redundant. However, models like OpenCL that can perform well on CPUs, GPUs *and* devices such as the Intel Xeon Phi KNC are demonstrating their flexibility to novel architectures. It is proposed that this could present an important predictor of long term success as the technologies that will support exa-scale computation may well seem very novel compared to the current most used devices [14].

10.7 BANDWIDTH

As the TeaLeaf application contains memory bandwidth algorithms, plotting the memory bandwidth achieved by each model presents a clear quantification of how much a model is taking advantage of a device's key resource.

A manual review of the whole application collected a close approximation of the number of bytes read and written while solving a problem with the CG solver. As discussed in subsection 8.3.1, there are at least two opposing ways to approach this task, providing upper and lower bounds on the bandwidth achieved by an application. The upper bound assumes that only contiguous memory accesses are free, and counts reads and write individually, but the lower bound assumes that each array element is accessed exactly once.

This analysis was performed for the CG solver at mesh convergence, and the disparity between the upper and lower bound (the hatched segment), shows just how uncertain the bandwidth calculations are in the absence of a method of direct measurement. It is important to recognise, however, that while the absolute values are not exact, the relative order of the models is because the same calculation has been performed for all models. Analysing the data suggests that the true value is likely closed to the lower bound, because the upper bound for CUDA actually exceeds the maximum potential bandwidth of the device, making it less convincing.

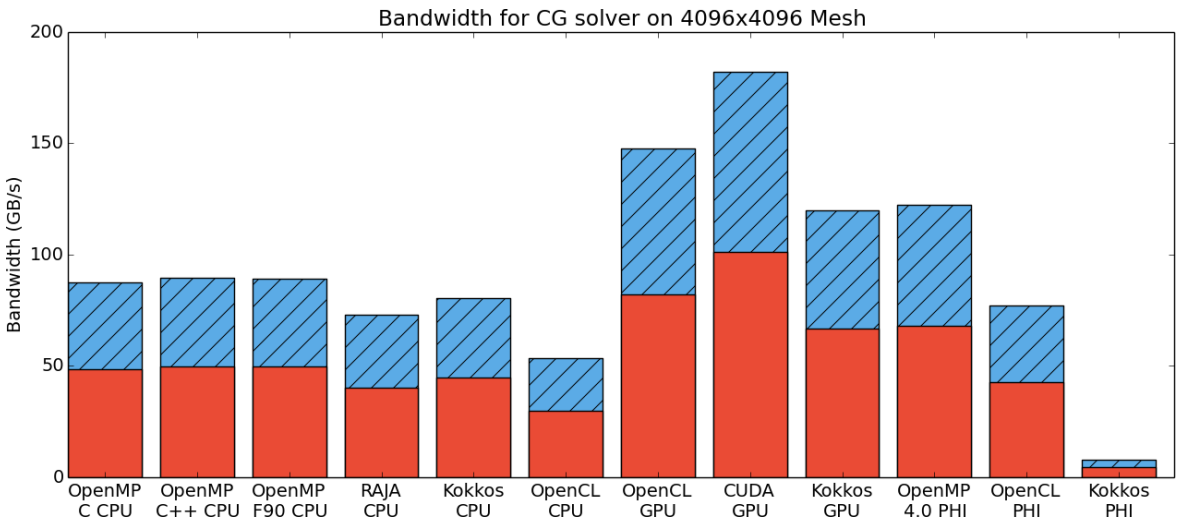


Figure 40: Bandwidth exhibited for the CG solver across all implementations.

Figure 40 does not take into account the peak of each device, which McIntosh-Smith et al. [33] suggest affords a better comparative evaluation of the performance portability of a model. Figure 41 takes the lower bound of bandwidth calculated for each model and shows the percentage of memory bandwidth relative to the benchmarked peak device bandwidth presented in section 8.1, as discussed in chapter 5. As this result uses the lower bound, it is important to recognise that the actual bandwidth utilisation will be slightly higher for all devices uniformly.

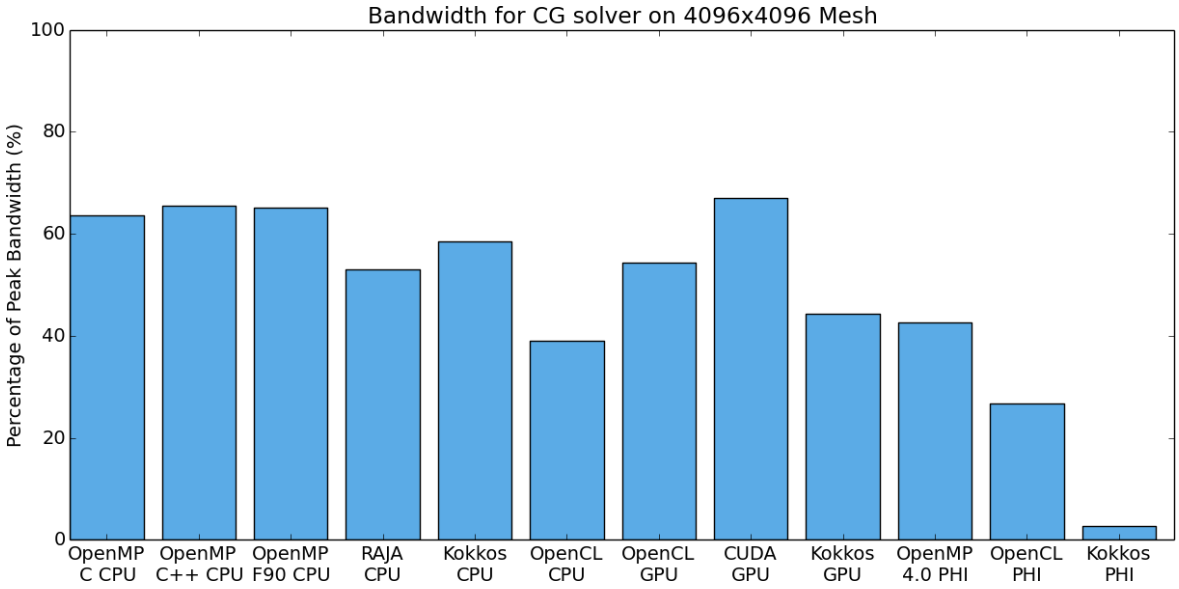


Figure 41: Bandwidth for each model relative to the peak attainable bandwidth on device.

While the GPU devices demonstrated the highest bandwidth utilisation, it can be seen in Figure 41 that the device-tuned models all achieve roughly 60% of device memory bandwidth. Although the OpenMP 4.0 implementation is intended to represent a performance portable implementation, it actually achieves by far the best memory bandwidth utilisation on the Intel Xeon Phi KNC.

The other performance portable models all fall slightly short of optimal utilisation, with the closest ports being Kokkos and RAJA on the CPU, and OpenCL on the GPU. OpenCL on the CPU, Kokkos on the GPU, and OpenCL on the Intel Xeon Phi KNC all suffer more than a 30% bandwidth penalty, and the estimated Kokkos value on the Intel Xeon Phi KNC represents by far the worst bandwidth.

OpenCL is the most balanced of all of the models, and while the model achieves significantly less bandwidth on the CPU, the CG solver appears to suffer from an unknown issue that may well be optimisable if discovered.

10.7.1 *Conclusions*

Analysing the bandwidth offers a very important and modern approach [33] to understanding the performance of the models across all of the tested devices. By standardising the results to be relative to the peak memory bandwidth of each device, the models are given more fair representation and it is clear exactly which models are able to take advantage of each devices resources. It is important to recognise that the Intel Xeon Phi KNC devices only achieve up to 40% of peak bandwidth, which corroborates a known deficiency with the architecture [43, 12, 15].

10.8 DISCUSSION

At this point it is important to consider the quantitative and qualitative findings together, in order to comment upon the overall success of each model.

The findings have given a mixed review for the performance portable implementations. In general, the device-specific versions demonstrate the best performance and provide the most stable results. If the absolute maximum performance is required then, unsurprisingly, this research suggests it would be best to use OpenMP (in particular Fortran) for the CPU, CUDA for NVIDIA GPUs and OpenMP 4.0 for Intel Xeon Phi KNC devices.

It is clear that at mesh convergence, both Kokkos and OpenCL are able to work across the full range of devices, demonstrating good functional portability. It is important to note that Kokkos is limited beyond this point, and cannot target AMD GPUs for instance, and RAJA is only able to target CPUs at the time of this investigation.

RAJA encompasses a very new concept and lacks maturity and extensive implementation, but the current results on the CPU demonstrate that the concept is heading in the right direction. The design approach resulted in a very clean source code, but pre-caching the indirection arrays pushed all of the core loops into mesh generation. In TeaLeaf's case many of the loops simply iterated over the domain excluding ghost cells, reducting duplication, but developing more complicated applications would require care to avoid an explosion of loops generating indirection arrays. Once more implementations are available it should be easy to switch between them using the generic policy approach and RAJA can be judged again.

For a performance portable model to achieve within 20% performance on both a CPU and NVIDIA GPU is quite tolerable, and shows that Kokkos already exhibits reasonable performance portability. The Kokkos result on the Intel Xeon Phi KNC is not a major concern because the KNC archiecture is being replaced with a new coprocessor design in the long term. The Kokkos port required a reasonable understanding of C++ templates and a very deep re-architecture of any source that was not already using templates, which rep-

resents a significant development effort. The final application, however, is reasonably performance portable, well architected, and easy to extend. The long term success of Kokkos relies upon the framework maturing and being extended to support new devices.

The OpenMP 4.0 development took much longer than originally anticipated, given the ease of developing OpenMP for the CPU, and required some architectural changes to TeaLeaf even though it is a relatively small benchmark. The performance achieved by the model was the best on the Intel Xeon Phi KNC, but the framework could not be considered performance portable or even predicted to be due to a lack of functional portability and issues with the way that data transfers are tied to lexically structured scopes.

OpenCL is often marginally slower than the other implementations, but this is most definitely offset by the functional portability possible with the standard. It can target a massive range of devices by changing simple flags, but does require specialist knowledge to develop, and ensure performance portability for those devices. Also, the variability issues discovered for the Intel CPU implementation are of some concern and would need to be resolved in TeaLeaf for the port to be viable.

Although the result is already well understood, it has been shown that for problems like TeaLeaf, it is essential to utilise devices that have a high memory bandwidth. At mesh convergence both the Intel Xeon Phi KNC and NVIDIA K20 were able to achieve good performance and this could be enabled with CUDA, OpenMP 4.0, Kokkos or OpenCL. All of the performance portable models have presented several outlying cases, which will need further investigation but are perhaps indicative of the additional complexity required for portable implementations.

11

CRITICAL EVALUATION

The main aim of this project was to present a fair and unbiased experimental investigation into the merits of a range of parallel programming models using the TeaLeaf mini-app. There were a number of milestones that had to be accomplished during the project, each of which is discussed independently.

11.1 DESIGN, DEVELOPMENT AND CONFIGURATION

The original plan was to output three new implementations of TeaLeaf in 3D: (1) CUDA, (2) RAJA or OpenMP 4.0, and (3) Kokkos. By the end of the project, all considered ports had been created to support testing: CUDA, RAJA, OpenMP 4.0, Kokkos, and a redevelopment of the Fortran OpenMP implementation in C and C++. The research project has also output 2D and 3D implementations of each application. Overall, the complete set of applications included 12 independent ports, some having several configurations for different devices, which represented a significant effort to develop and maintain throughout testing.

Up-front design for each port was relatively limited, given that most of the technologies were very new and lacked documentation or example codes. In general, the migration process for each technology was complex and time consuming, but once the original design had been implemented the final optimised source was often only subtly different. OpenMP 4.0 on the other hand required many architectural revisions to achieve the maximum attainable performance, and the up-front design proved to be completely deficient. Having only had experience with OpenMP (and not 4.0), expertise had to be achieved on the fly and the designs were vague through a lack of working knowledge with the technologies. The results show that this challenging process was successful as the full range of ports were implemented and each can capably handle the TeaLeaf feature set.

To support the migration process, the original Fortran OpenMP implementation was used to develop a functionally identical C++ implementation of the computation kernels. By creating this implementation there were two significant benefits (1) the OpenMP C++ implementation could be tested alongside the other ports, and (2) CUDA, Kokkos, RAJA, OpenMP C and OpenMP 4.0 all derived from this tested and proven application. Had this step been omitted there would more than likely have been a disparity between the implementations, which could have slowed down development and introduced experimental bias. Also, in order to ensure valid testing, a number of constraints were outlined that supported developing applications without introducing accidental bias, and it is believed that this can give greater confidence in the results.

The OpenCL implementation of TeaLeaf was created prior to this research project but the open source 2D implementation had undergone significant revisions since the 3D implementation, trialling experimental improvements to the computational kernels. Including such improvements in a single application would have surely introduced bias and so a significant amount of time was expended creating a new 2D implementation that identically matched the 3D feature set. Unfortunately this represents work that is entirely redundant beyond this research project, but was necessary to ensure maximum experimental validity.

Results were collected for all of the applications, including the 2D and 3D implementations, but they were very similar and so the results for the 3D ports were excluded from this report for brevity. While this effort provided little visible benefit to the research project in isolation, the 2D *and* 3D ports are complementary and represent an important and unique set of benchmarks that are already being utilised in research. For instance, at the time of writing, the OpenMP 4.0 implementation is being used to test a beta implementation of the PGI compiler suite for The Portland Group. Overall, the implementation phase of the project was far more successful than expected and, other than the difficulties with OpenMP 4.0 and OpenCL, suffered from few inhibitors.

11.2 TESTING AND VISUALISATION

Of all the activities performed in this project, testing spanned the greatest total time but was mostly accomplished in parallel with other tasks. Drawing from prior experience it was known that up-front work on automation and test design has a major influence on the long term efficiency and ultimate success of such processes. Developing timing collection and profiling modules meant that data could be collected with minimal manual involvement on the supercomputer. Although this strategy had a significant up-front cost, it was clear that without such a thorough and considered test approach it would not have been possible to collect such a wealth of quality data whilst also guaranteeing its validity.

Custom log file parsing was used to handle the timing aspects and this worked very capably, being lightly added to each implementation without affecting the core code at all. On reflection it may have been better to have tied the timing files to a specific configuration somehow, as this would have completely removed the manual processing post-optimisation.

Another important aspect of the testing was that the Blue Crystal phase 3 supercomputer is in constant use by other researchers. This means that jobs would often have to complete over-night, greatly emphasising the importance that the jobs were successful. Initially, many failures were experienced with jobs, and it was essential at this stage to re-evaluate the process and improve the job submission process. A careful reconsideration resulted in individual job submission scripts for each application and test, which were carefully standardised before continuing, greatly improving the success rate.

Throughout the project it was often difficult to balance exactly which tasks or tests should take priority to maximise downstream efficiency. For instance, after some rounds of testing it was discovered that the 'KMP_AFFINITY' parameter makes a significant difference to the variance encountered during testing. Had this parameter been discovered earlier it may have reduced the number of iterations required to achieve a stable result, reducing the net effort. If this project were to be repeated, it would be far more effective to more carefully consider and test all available parameters prior to performing other testing.

Initially, some time was wasted in automating strong scaling experiments, which turned out to be irrelevant given that none of the technologies handled inter-node communication. To avoid this it may have been better to write the test automation after the set of ports were complete rather than at the same time. This is a difficult issue however, as the testing and implementation are most efficiently performed at the same, so that work can be overlapped and results validated immediately.

In the end, over 15,000 invocations of the TeaLeaf application were performed for a range of configurations and mesh sizes, spanning around 450 hours of solve time. Each time an optimisation was applied it was useful to repeat the tests and re-evaluate the

data, and repeating this process resulted in data that was smooth and representative of the optimised implementations.

11.3 OPTIMISATION AND RESULTS

Visualisation was an essential tool for effectively completing this research project, allowing masses of raw data to be easily interpreted to inform continual testing. The use of Python plotting capabilities proved to be a fantastic option for creating flexible scripts that could plot the raw data in a variety of styles.

The optimisation process exposed some interesting insights, for instance it is expected that discussing optimising OpenMP 4.0 in a non-trivial application presents a relatively rare resource for future developers and researchers. Deficiencies discovered with the OpenCL CPU implementation represent an important issue that demands investigation to avoid difficulties in production codes. Also, through targetted profiling it was possible to discover and remedy a false sharing problem in the RAJA alpha implementation, which is being fed back to the developers. These results are quite unique and represent some of the long term impacts that have extended from this project.

There were several results encountered that exhibited unusual performance characteristics but could not be explained even after significant investigation. Ideally those instances of strange behaviour would have all been eliminated to make the results as convincing as possible, however it is hoped that the analysis provided can support future investigation. If possible it would have been beneficial, although not essential, to gain unrestricted access to an Intel Xeon Phi KNC device, to calculate accurate performance for the native compilation at mesh convergence.

After careful tuning it was possible to present a compelling argument about the performance of the models at mesh convergence. Some results are shown for smaller mesh sizes, and this corroborates the anticipated result that fast CPU cache is highly effective for small data loads. However, it is believed that focussing upon mesh convergence, where possible, provides the most accurate representation of the performance of each model at the intensity required for real scientific problems. Having minimised experimental bias and collected quality data, the results and discussions clearly satisfy the original aims of the project, and several auxillary results add value beyond those aims.

11.4 CONCLUSION

The research project has successfully implemented an extensive set of ports using novel parallel programming models, and presented quantitative proof that uniquely compares and evaluates them. Ultimately, the research project has delivered above and beyond the originally intended outcomes, with contributions to the research community that offer genuine long term benefits. The results have been commended by the UoB HPC group and the industrial partner AWE, as they represent information not available elsewhere that will have an impact on the decision making for scientific application developers. The core results of the performance evaluations of Kokkos and RAJA are to be presented as an academic conference poster at the International Conference for High Performance Computing, Networking, Storage and Analysis 2015 (SC15), the draft is available in Appendix B.

12

CONCLUSIONS

12.1 FURTHER WORK

Throughout the project several unanticipated issues have been encountered, and future research could look deeper into the causes. For instance, there have been some isolated instances of poor performance exhibited by some models, such as the TeaLeaf Fortran PPCG and OpenCL CPU CG solvers. Given that the results are inconsistent between solver, mesh sizes and application, it will be important to understand whether those performance issues lie with the ports or the model implementations.

The Intel compiler suite is shown to greatly improve performance of OpenMP applications on Intel architecture, compared to the same code compiled with the GNU suite. Future research could investigate whether this performance difference is consistent on non-Intel hardware. As discussed in section 10.4, RAJA Chebyshev appears to preclude the Intel optimisation and, if this correlation valid, comparing the RAJA implementation to another model that was optimised well by the Intel compiler may help to expose which optimisations are being excluded.

When investigating the performance of the OpenMP implementation of TeaLeaf, a performance disparity was observed between the C, C++ and Fortran ports. It would be very useful to take this further in the future and understand exactly why compiling the C application with the C++ compiler would cause such a performance penalty.

Throughout the project, high variability was encountered for the performance results of the OpenCL CPU implementation. It is hypothesised that this is caused by the Intel Thread Building Blocks backend that underpins the Intel OpenCL implementation, and it would be important to determine whether this is definitely the case. As a starting point, the tests described in subsection 9.4.2 should be performed with older and/or newer versions of the Intel OpenCL implementation to see if this issue is specific to the version used in testing.

Further to this, the project has exposed some interesting opportunities for continuing research that extends the analysis presented.

- The TeaLeaf benchmark is memory bandwidth bound, with fairly simple data structures. It would be useful to complement this work with a compute bound problem and/or an application that requires more complicated data structures.
- Although data was collected for the 2D and 3D applications, it was impossible to fit all of the possible analyses into this project. As the data will be made publicly available, there is the potential for future research to revisit the performance results, for instance to better understand the differences between the performance profiles of the 2D and 3D applications.
- It was not possible to include comparisons on diverse devices but it may prove useful to observe the same tests run on a range of devices, including AMD CPUs and other NVIDIA GPUs. Once a Knights Landing architecture device is available it would also make an interesting comparison against the KNC results presented here.

12.2 SUMMARY

Throughout this project, a number of unbiased insights have been presented, some of which are believed to be entirely unique. It has been shown that modern performance portable programming models *are* able to support the development of applications that can take advantage heterogeneous devices, but all the models analysed require a trade-off between performance, development complexity, and functional portability. It has been shown that RAJA can competetively support CPU execution but needs to support more devices to be a truly useful model. Kokkos exhibits good performance portability between CPUs and NVIDIA GPUs, but the MIC native compilation is not performance portable and would need to be changed to an offload model to compete with other technologies.

OpenCL demonstrates the best functional portability and performed reasonably well on all tested devices, but generally suffered at least a 1-10% performance penalty compared to the other implementations. Also, the testing of OpenCL on the CPU was made much more difficult because of its inconsistent runtimes, potentially caused by Intel Thread Building Blocks. OpenMP 4.0, once optimised, can achieve the best performance on an Intel Xeon Phi KNC, but the lack of unstructured data persistence mades the model challenging to achieve good performance in TeaLeaf, which is a non-trivial application. The detailed explanation of the key techniques used to optimise the OpenMP 4.0 implementation should support developers in implementing OpenMP 4.0 in their applications.

Several auxillary findings were presented, for instance at the lowest level differences in performance were observed between C and Fortran and C++, that suggest C++ achieves worse performance for problem sizes that fit into CPU cache. Further to this, an important outcome of the results of this research project is that to handle computation where there is a large data load, such as at mesh convergence, GPUs offer far superior computational efficiency than CPUs. This has been clearly demonstrated in the data and corroborates the idea that for large memory-bound problems, high bandwidth devices are the best option.

Further to this, the research has discussed a number of interesting additional topics such as: the design and development of each port, how to perform experiments with multiple implementations at once, and optimising the models for heterogeneous execution. Importantly, the released applications will be highly valuable research tools that can support industry and academia in future research.

As the outcomes of the project are numerous, generating a considerable volume of software, and unique qualitative and quantitative results, there is still a great amount of potential for the project to continue. All of the source code and raw performance data collected as part of this research project will be made publically available through the UK Mini-App Consortium [52] repository to support continuing research.

Part IV

APPENDIX

A

ADDITIONAL RESULTS

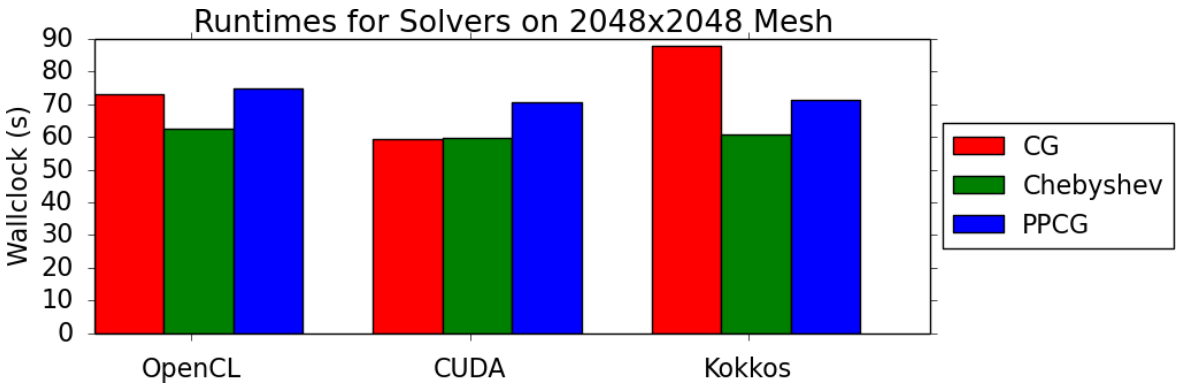


Figure 42: Results of applicable models on the **NVIDIA K20 GPU**.

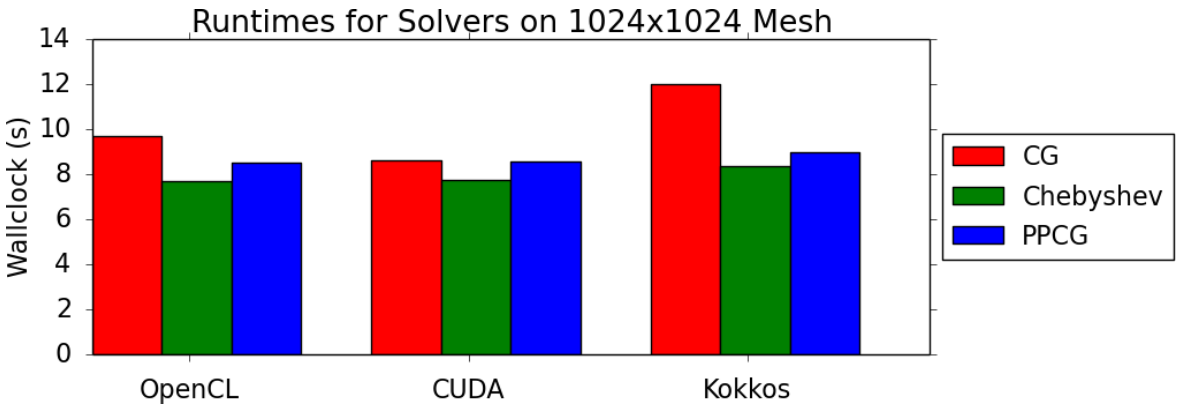


Figure 43: Results of applicable models on the **NVIDIA K20 GPU**.

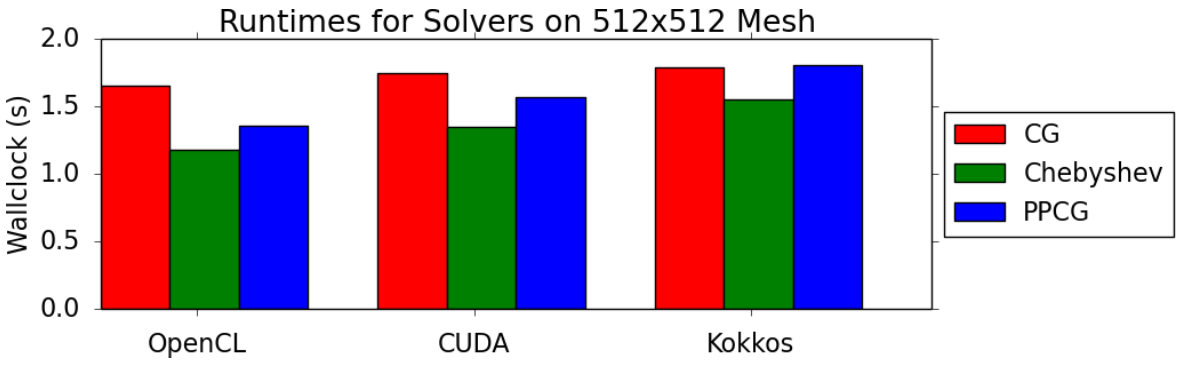


Figure 44: Results of applicable models on the **NVIDIA K20 GPU**.

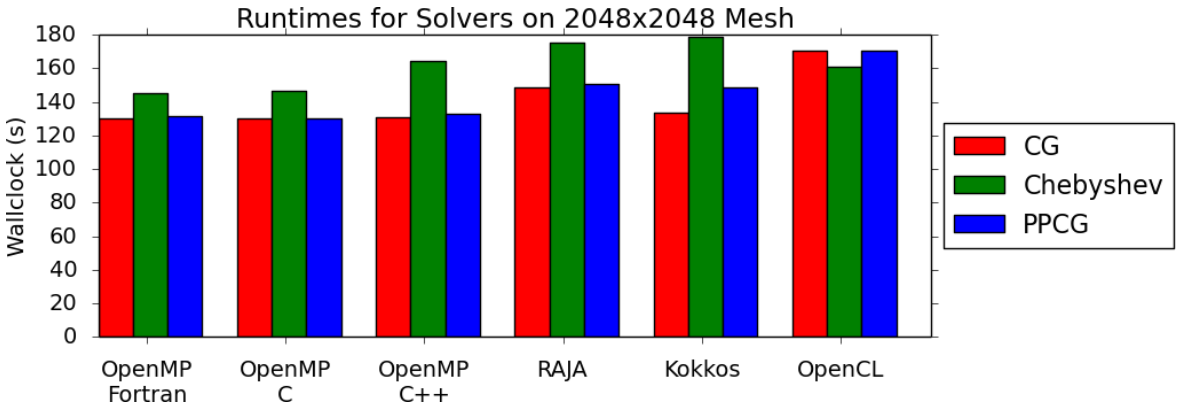


Figure 45: Results of applicable models on 2 **Intel Xeon E5-2670 CPUs**.

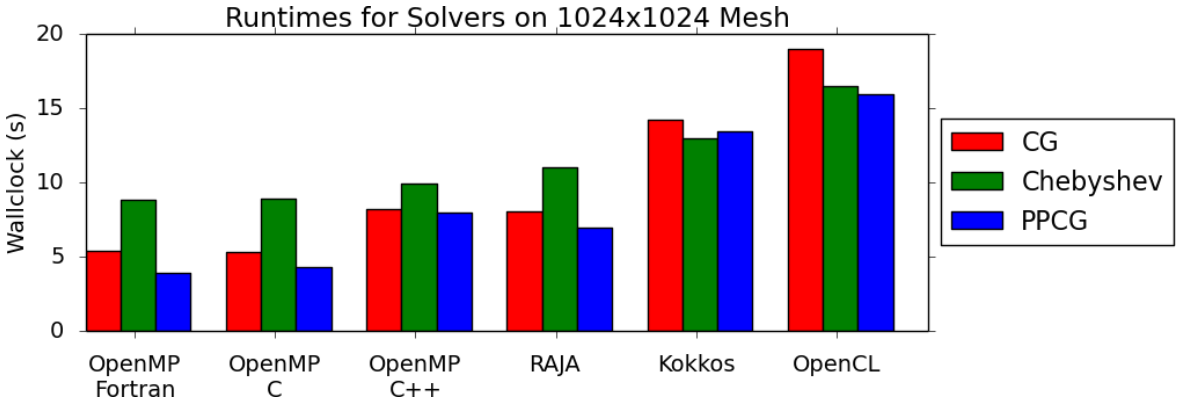


Figure 46: Results of applicable models on 2 **Intel Xeon E5-2670 CPUs**.

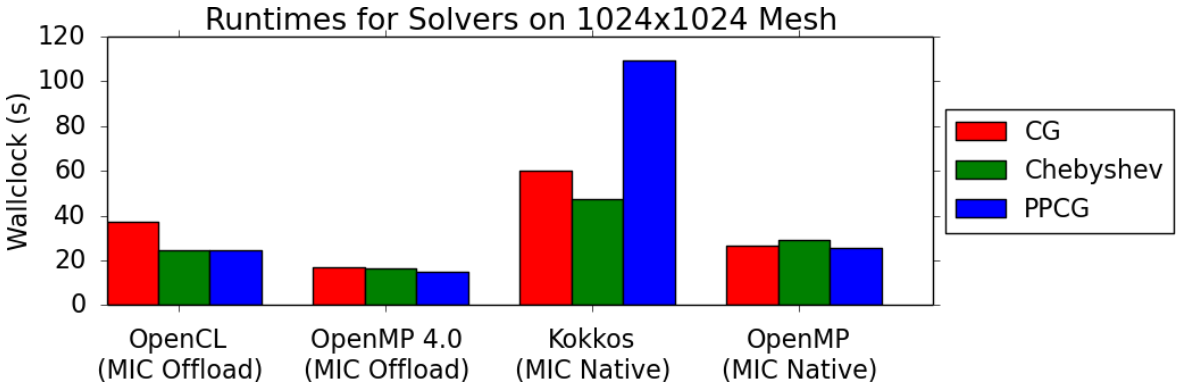


Figure 47: Results of applicable models on the **Intel Xeon Phi KNC**.

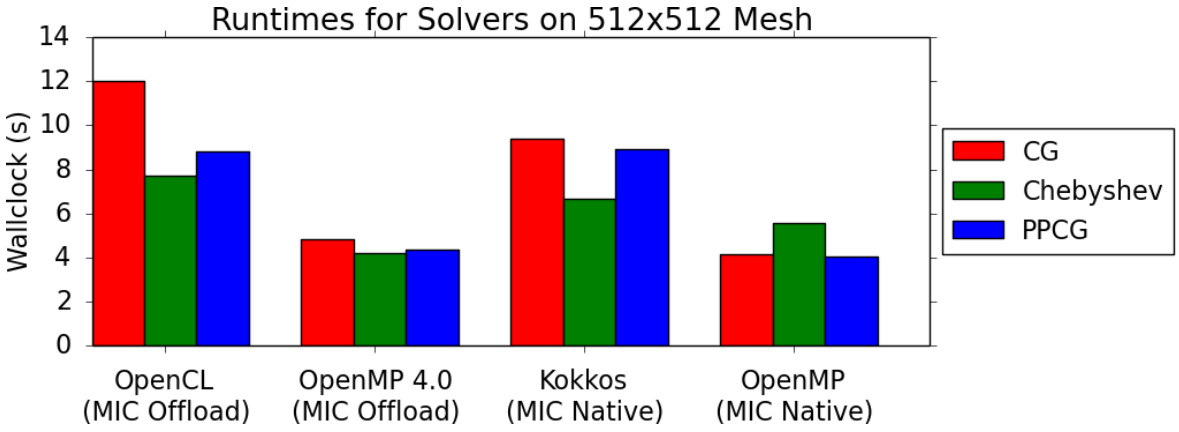


Figure 48: Results of applicable models on the **Intel Xeon Phi KNC**.

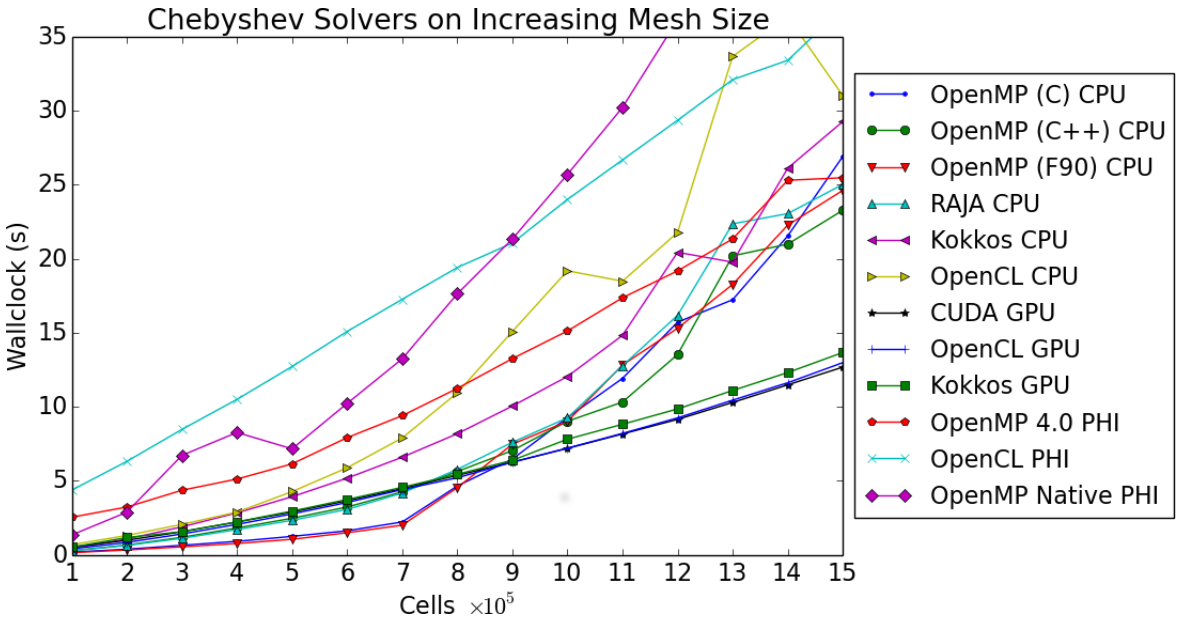


Figure 49: Results of evenstep experiment for **Chebyshev** solver.

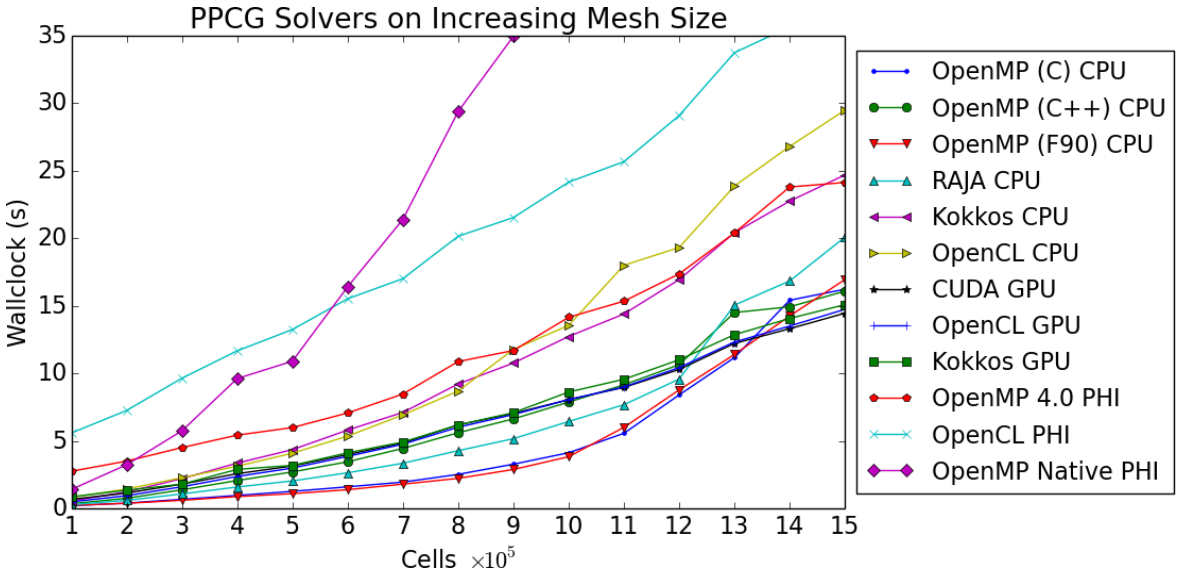


Figure 50: Results of evenstep experiment for **PPCG** solver.

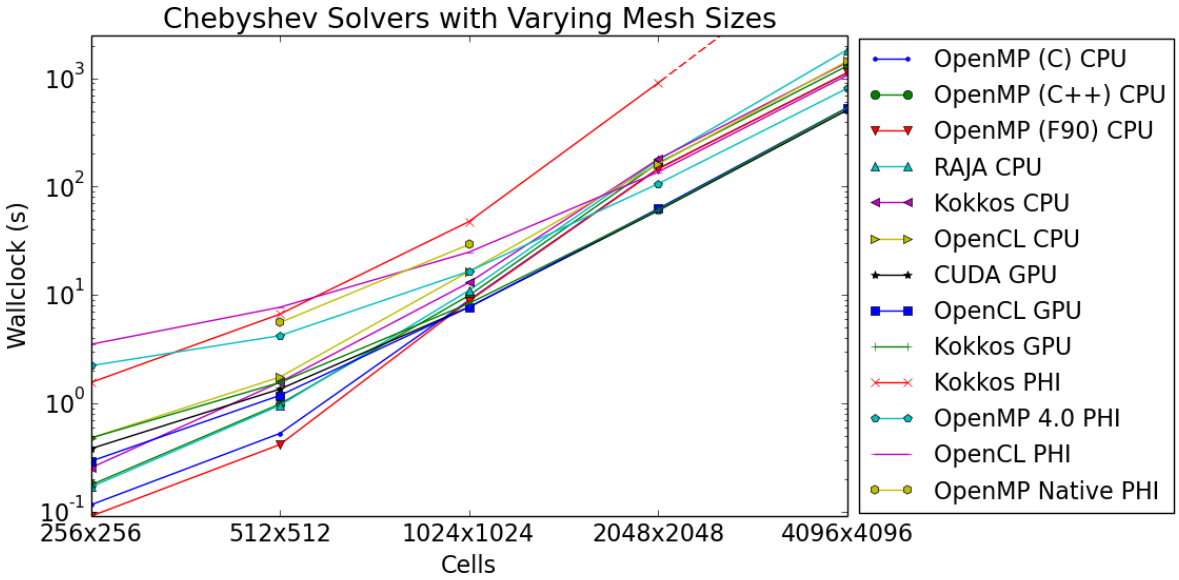


Figure 51: Results of power of 2 experiment for **Chebyshev** solver.

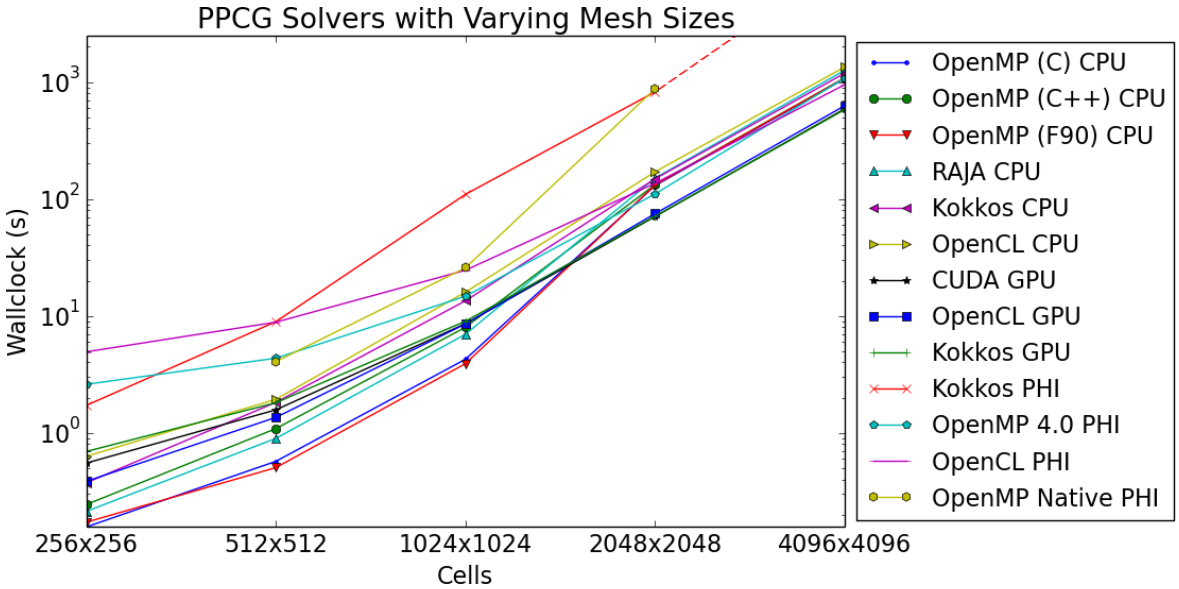


Figure 52: Results of power of 2 experiment for **PPCG** solver.

B

TEALEAF CODE AND SC15 POSTER

```
// Entry point for initialising sd
extern "C"
void ext_ppcg_init_sd_(
        const int* chunk, double* alphas, double* betas, int* maxChebyIters)
{
    auto c = Chunks[*chunk-1];

    c->alphas = View<double*,DEVICE>("alphas",*maxChebyIters);
    KokkosHelper::InitView<double>(c->alphas,alphas,*maxChebyIters);

    c->betas = View<double*,DEVICE>("betas",*maxChebyIters);
    KokkosHelper::InitView<double>(c->betas,betas,*maxChebyIters);

    PPCGInitSd<DEVICE> kernel(c->dims, c->sd, c->r, c->theta);

    START_PROFILING;
    Kokkos::parallel_reduce(c->fullDomain,kernel);
    STOP_PROFILING("PPCG Init SD");
}

// Initialises sd
template <class Device>
struct PPCGInitSd
{
    typedef Device device_type;

    PPCGInitSd(TLDims dims, KView sd, KView r, double theta)
        : dims(dims), sd(sd), r(r), theta(theta){}

    KOKKOS_INLINE_FUNCTION
    void operator()(const int index) const {
        KOKKOS_INDICES;

        if(INDEX_IN_INNER_DOMAIN)
        {
            sd[index] = r[index] / theta;
        }
    }

    TLDims dims;
    Kokkos::View<double*,Device> sd;
    Kokkos::View<double*,Device> r;
    double theta;
};
```

Figure 53: A representative sample of code from the Kokkos TeaLeaf port.

```
// Updates the front facing halo
void TeaLeafChunk::UpdateFront(const int depth, double* buffer)
{
#pragma omp parallel for
    for(int ii = 0; ii < depth; ++ii)
    {
        for(int jj = HALO_PAD; jj < yCells - HALO_PAD; ++jj)
        {
            for(int kk = HALO_PAD; kk < xCells - HALO_PAD; ++kk)
            {
                int base = jj * xCells + kk;
                buffer[base + (zCells - HALO_PAD + ii) * page]
                    = buffer[base + (zCells - HALO_PAD - 1 - ii) * page];
            }
        }
    }
}
```

Figure 54: A representative sample of code from the OpenMP C++ TeaLeaf port.

```
// Calculates the new value of w
void TeaLeafChunk::CGCalcW(double* pw)
{
    *pw = 0.0;

    START_PROFILING;
    forall_sum_inner<policy>(*rajaHelper->InnerDomainList, pw,
        [&] (int index, double* pwTemp) {
            w[index] =  (1.0 + (kx[index+1]+kx[index])
                + (ky[index+xCells]+ky[index]))*p[index]
                - (kx[index+1]*p[index+1]+kx[index]*p[index-1])
                - (ky[index+xCells]*p[index+xCells]
                    +ky[index]*p[index-xCells]);
            *pwTemp += w[index]*p[index];
    });
    STOP_PROFILING("CG Calc W");
}
```

Figure 55: A representative sample of code from the RAJA TeaLeaf port.

```
// Initialises the Chebyshev solver
void TeaLeafCudaChunk::ChebyInit(
        const double* alphas, const double* betas, int numCoefs,
        const double theta)
{
    // Load alphas and betas to device
    const int length = numCoefs*sizeof(double);
    cudaMalloc((void**) &dAlphas, length);
    cudaMalloc((void**) &dBetas, length);
    cudaMemcpy(dAlphas, alphas, length, cudaMemcpyHostToDevice);
    cudaMemcpy(dBetas, betas, length, cudaMemcpyHostToDevice);
    CheckErrors(__LINE__,__FILE__);

    const int innerX = xCells - 2 * HALO_PAD;
    const int innerY = yCells - 2 * HALO_PAD;
    const int numThreads = innerX * innerY;
    const int numBlocks = std::ceil(numThreads / (float)BLOCK_SIZE);

    START_PROFILING();
    CuKnlChebyInit<<<numBlocks, BLOCK_SIZE>>>(
            innerX, innerY, xCells, dU, dU0, dKx, dKy, theta, dP, dR, dW);
    STOP_PROFILING("Cheby Init");
    CheckErrors(__LINE__,__FILE__);
}

// Kernel that initialises Chebyshev solver
__global__ void CuKnlChebyInit(
        const int innerX, const int innerY, const int xMax,
        const double* u, const double* u0, const double* Kx,
        const double* Ky, const double theta, double* p, double* r,
        double* w)
{
    const int gid = threadIdx.x + blockIdx.x * blockDim.x;
    if(gid >= innerX * innerY) return;

    const int col = gid % innerX;
    const int row = gid / innerX;
    const int off = HALO_PAD * (xMax + 1);
    const int index = off + col + row * xMax;

    const double smvp = (1.0 + (Kx[index+1]+Kx[index])
            + (Ky[index+xMax]+Ky[index]))*u[index]
            - (Kx[index+1]*u[index+1]+Kx[index]*u[index-1])
            - (Ky[index+xMax]*u[index+xMax]+Ky[index]*u[index-xMax]);

    w[index] = smvp;
    r[index] = u0[index] - w[index];
    p[index] = r[index] / theta;
}
```

Figure 56: A representative sample of code from the CUDA TeaLeaf port.

tea_solve.f90:

```
!$OMP DECLARE TARGET(timer, OMP_GET_THREAD_NUM, ext_cg_solver_init, ext_solver_copy_u)
!$OMP DECLARE TARGET(ext_calculate_2norm, ext_calculate_residual, ext_cheby_solver_iterate)
!$OMP DECLARE TARGET(ext_cg_calc_w, ext_cg_calc_ur, ext_cg_calc_p, ext_solver_finalise)
!$OMP DECLARE TARGET(ext_jacobi_kernel_init,ext_jacobi_kernel_solve)

IF(task.EQ.parallel%task) THEN
    !$OMP TARGET UPDATE TO(max_iters, g_out, parallel, profiler_on, profiler, dt, dx, dy, chunks)
    !$OMP TARGET UPDATE TO(use_ext_kernels, tl_use_chebyshev, tl_use_cg, tl_use_ppcg, tl_use_jacobi)
    !$OMP TARGET UPDATE TO(use_fortran_kernels, eps, coefficient, tl_ch_cg_errswitch, tl_ch_cg_epslim)
    !$OMP TARGET UPDATE TO(tl_ch_cg_presteps, tl_check_result, tl_ppcg_inner_steps, tl_preconditioner_on)
    !$OMP TARGET UPDATE TO(complete)

    !$OMP TARGET MAP(tofrom: density, energy1, energy0, u, u0, cg_alphas, cg_betas, ch_alphas, ch_betas) &
    !$OMP MAP(tofrom: chunk_neighbours, fields, n, r, sd, Kx, Ky, Mi, w, p, z)

    DO n=1,max_iters
        ...

        CALL ext_cg_calc_p(c, p, r, beta)

        ...
    ENDDO

ENDIF
```

ext_cg_solver.c:

```
#pragma omp declare target

// Entry point for calculating p
void ext_cg_calc_p_(const int* chunk, double* p, double* r, const double* beta)
{
    START_PROFILING;
#pragma omp parallel for
    for(int jj = HALO_PAD; jj < _chunk.y - HALO_PAD; ++jj)
    {
        for(int kk = HALO_PAD; kk < _chunk.x - HALO_PAD; ++kk)
        {
            const int index = jj * _chunk.x + kk;
            p[index] = (*beta) * p[index] + r[index];
        }
    }
    STOP_PROFILING("CG Calc P");
}

#pragma omp end declare target
```

Figure 57: A representative sample of code from the OpenMP 4.0 TeaLeaf port.

```
#!/bin/bash
#PBS -N tealeaf_openmp4
#PBS -l nodes=1:ppn=16:xeon-phi,walltime=01:00:00
#PBS -q testq

np=1
app="./tea_leaf"

cd $PBS_O_WORKDIR
cat $PBS_NODEFILE > machine.file.$PBS_JOBID

export KMP_AFFINITY=compact
export OMP_NUM_THREADS=16

sizes=(316 447 548 632 707 775 837 894 949 1000 1049 1095 1140 1183 1224)
for i in "${sizes[@]}"; do
   rand=$RANDOM
   mpirun --machinefile machine.file.$PBS_JOBID -np $np $app \
       "../configs/tea.in.$i.cg" "../out_files/out.openmp4.cg.$rand" "../timing.evenstep"
   mpirun --machinefile machine.file.$PBS_JOBID -np $np $app \
       "../configs/tea.in.$i.chebyshev" "../out_files/out.openmp4.chebyshev.$rand" "../timing.evenstep"
   mpirun --machinefile machine.file.$PBS_JOBID -np $np $app \
       "../configs/tea.in.$i.ppcg" "../out_files/out.openmp4.ppcg.$rand" "../timing.evenstep"
done
```

Figure 58: A representative sample of a PBS job submission script.

ext_cuda_chunk.cpp

```
TeaLeafCudaChunk::~TeaLeafCudaChunk()
{
#ifdef ENABLE_PROFILING
    int N = xCells-HALO_PAD*2;
    int M = yCells-HALO_PAD*2;

    // Static list of bytes read/written by function
    map<string, int> functions;
    functions["CG Init U"] = (2*(N+4)*(M+4)+3*(N+4)*(M+4));
    functions["CG Init W"] = ((N+2)*(M+2)+(N+2)*(M+2));
    functions["CG Init K"] = (2*(N+1)*(M+1)+2*(N+1)*(M+1));
    functions["Finalise"] = (2*N*M+N*M);

    PRINT_PROFILING_RESULTS(cudaProfileOut, functions);
#endif
}
```

ext_profiler.hpp:

```
#define PRINT_PROFILING_RESULTS(outfile,functions) \
    ExtProfiler::Instance().PrintResults(outfile,functions);
```

ext_profiler.cpp

```
void ExtProfiler::PrintResults(string outfile, map<string, int> functions)
{
    ostringstream oss;
    oss << setiosflags(ios::fixed) << left << endl
        << setw(30) << "Kernel Name" << setw(15) << "Time (ms)"
        << setw(10) << "Calls" << setw(10) << "ms/call"
        << setw(15) << "Bandwidth" << endl;

    double total = 0.0;
    typedef map<string, profile>::iterator it_type;
    for(it_type p = profiles.begin(); p != profiles.end(); p++)
    {
        double timeInSecs = p->second.time / MS;
        double bandwidth = (sizeof(double) * p->second.calls)
            * (functions[p->first] / (timeInSecs*GB);
        total += p->second.time;
        oss << setw(30) << p->first
            << setw(15) << setprecision(3) << p->second.time
            << setw(10) << p->second.calls
            << setw(10) << p->second.time/p->second.calls
            << setw(15) << bandwidth << endl;
    }

    oss << "Total kernel time: " << total << "ms" << endl << endl;

    ofstream out(outfile.c_str(), ios::app);
    out << oss.str() << endl;
}
```

Figure 59: A representative sample of code from the custom profiling component.

```
# Filters the runs, and plots the result of particular test
def PlotResults(runs):

    # Filter runs
    ...

    # Plot results for CG solver
    fig = pyplot.figure(figsize=(12, 7))
    ax = pyplot.subplot(111)

    ticks=range(0, maxthreads)
    for rr in range(0, len(results)):
        ax.plot(ticks, results[rr].times, label=results[rr].version, marker=markers[rr])

    ax.set_ylim([0,30])
    ax.set_xlim([1,15])

    pyplot.xticks(range(0,maxthreads), threads)
    pyplot.title('CG Solvers on ' + _probsize + ' Mesh')
    pyplot.xlabel('Threads')
    pyplot.ylabel('Wallclock (s)')
    pyplot.show()
```

Figure 60: A representative sample of a graph script.

tea_leaf.f90

```
PROGRAM tea_leaf

  USE tea_module

  IMPLICIT NONE

  CHARACTER(len=g_len_max) :: tea_out, tea_in, out_log

! Prepare communications etc.
  ...

  CALL initialise(tea_in, tea_out)

  CALL diffuse

  CALL ext_finalise

! Enable logging simple timing details to a global log file
#ifdef ENABLE_TIMELOGGING
  out_log = g_out_log
  IF(iargc() >= 3) THEN
      CALL getarg(3,out_log)
  ENDIF
  IF(parallel%boss) THEN
      CALL ext_log_timings(out_log, tea_out)
  ENDIF
#endif

END PROGRAM tea_leaf
```

ext_timings.cpp

```
extern "C"
void ext_log_timings_(char* outLog, char* teaOutPath)
{
    string outLogClean = get_clean_string(outLog);
    string teaOutPathClean = get_clean_string(teaOutPath);

    ifstream teaOutFile(teaOutPathClean.c_str());

    if(!teaOutFile.is_open())
    {
        cerr << "Could not open " << teaOutPathClean << " to parse results." << endl;
        return;
    }

    string line;
    ostringstream oss;
    oss << get_time() << " " << get_num_threads();

    // Get the wallclock result
    while(getline(teaOutFile, line))
    {
        size_t pl = timePrefix.length();
        if(line.length() > pl && line.substr(0,pl) == timePrefix)
        {
            string timeWord = line.substr(pl);
            double time = atof(timeWord.c_str());
            oss << " " << setprecision(12) << time;
            break;
        }
    }

    ...

    ofstream log(outLogClean.c_str(), ios::app);
    log << oss.str() << endl;
}
```

Figure 61: A representative sample of non-invasive timing module.

SC15 Poster Submission: A Performance Evaluation of Kokkos & RAJA using the TeaLeaf Mini-App

Matthew Martineau*, Simon McIntosh-Smith*, Mike Boulton*, Wayne Gaudin†, David Beckingsale§
*Dept. of Computer Science, University of Bristol (cssnmis@bristol.ac.uk), †AWE, §LLNL

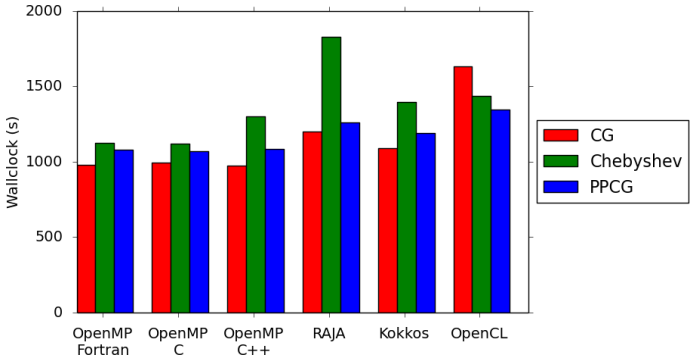
Introduction

TeaLeaf is an open source heat conduction 'mini-app', that is part of the Mantevo benchmark suite from Sandia [1]. It is designed to enable HPC experiments free from the burden of using a fully functional scientific program. The application is memory bandwidth bound and hosts three double precision iterative sparse matrix solvers: Conjugate Gradient (CG), Chebyshev, and Preconditioned Polynomial CG (PPCG). Through extensive testing of a number of new TeaLeaf 2D ports we have been able to demonstrate the performance profile of a range of diverse programming models across modern HPC devices. In particular this research focussed upon two new programming models: Kokkos (Sandia National Laboratories) and RAJA (Lawrence Livermore National Laboratories), which leverage template meta-programming and lambda functions respectively, to improve ease of development and long term functional portability. We show here that TeaLeaf RAJA demonstrates promising performance on CPUs and TeaLeaf Kokkos is competetively performance portable.

TeaLeaf 2D Performance Results

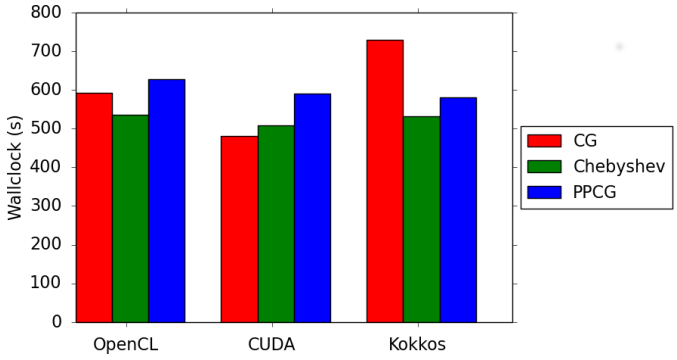
RAJA is only currently implemented for CPUs, but Kokkos can execute on CPUs, NVIDIA GPUs and supports native compilation for the Intel Xeon Phi Knights Corner. The net effort required to complete each port varied significantly, with RAJA and OpenMP allowing the fastest development time. The ports were compiled using the Intel Compilers (version 15.0), and performance testing was completed on the Blue Crystal supercomputer at the University of Bristol. Core logic and configurations were conserved across ports to reduce experimental bias. Where possible, the performance results are presented for the 4096x4096 mesh size, as this represents the point of mesh convergence.

Wallclock on Intel Xeon E5-2670 (Sandybridge) for 4096x4096 Mesh



Kokkos and RAJA demonstrate only a small performance penalty (less than 15% over the baseline OpenMP), and OpenMP outperforms all other models by between 5% and 40%.

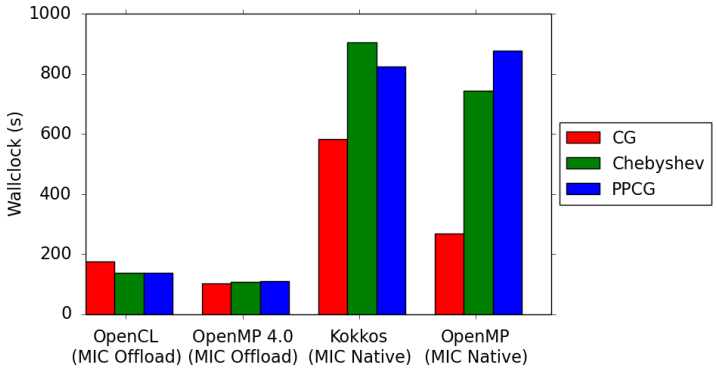
Wallclock on NVIDIA K20 for 4096x4096 Mesh



Kokkos exhibits impressive GPU performance for the Chebyshev and PPCG solvers, emitting CUDA (version 6.0.37) kernels that perform as well as our hand optimised implementation.

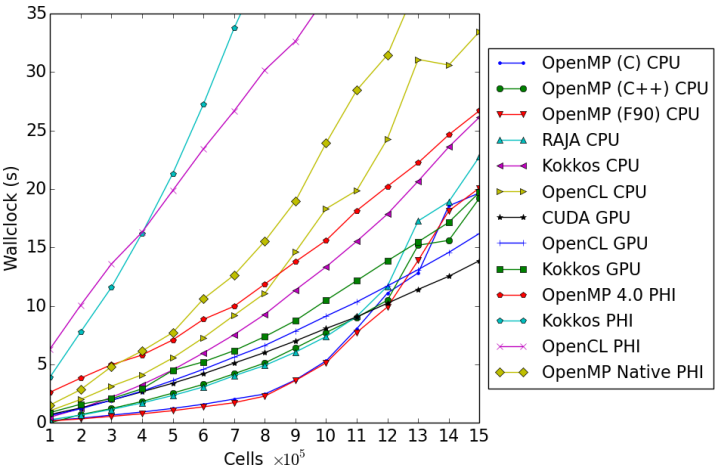
Whilst Kokkos requires radical re-structuring of C or Fortran codes, the abstractions shield the developer from some of the complexities inherent with developing CUDA. With a simple configuration change Kokkos is capable of running natively on accelerator cards.

Wallclock on Intel Xeon Phi SC5110P for 2048x2048 Mesh



Both the Kokkos and OpenMP native ports demonstrate similar poor performance on the accelerator. In contrast, the offloading models achieve good performance and, in particular, OpenMP 4.0 is significantly more performant than the other three implementations.

Wallclock as problem size increases in steps of 100,000



The data demonstrates that the OpenMP ports, including RAJA, dominate performance up until the point that CPU fast cache is saturated and the high bandwidth devices begin to take over.

Conclusion

TeaLeaf is a valuable research tool that has been used to successfully investigate the performance of a number of modern parallel programming models. We have shown that RAJA is a competetive model on the CPU, but it relies upon C++11 and requires a similar net effort to develop as OpenMP, and so must mature to support new devices to prove useful. Kokkos requires much greater up-front development cost and requires knowledge of C++ templates, but clearly enables good CPU performance and impressive GPU performance from the same codebase. For the accelerators, Kokkos suffers from poor performance, and cannot currently compete with OpenCL or OpenMP 4.0.

University of BRISTOL

UOB HPC

[1] Sandia National Laboratory: The Mantevo project home page. http://mantevo.org/

[2] M. Boulton, S. McIntosh-Smith: Optimising sparse iterative solvers for many-core computer architectures. UK Many-Core Developer Conference (UKMAC), Dec. 2014.

[3] UK Mini-App Consortium (UK-MAC) TeaLeaf page: http://uk-mac.github.io/TeaLeaf/

BIBLIOGRAPHY

[1] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[2] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] M. Barth, B. Byckling, N. Ilieva, et al. Best Practice Guide Intel Xeon Phi v1.1. http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/, 2014.

[4] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0, 2013.

[5] J.M. Cecilia, J.M. Garcia, and M. Ujaldon. CUDA 2D Stencil Computations for the Jacobi Method. In *Applied Parallel and Scientific Computing*, volume 7133 of *Lecture Notes in Computer Science*, pages 173–183. Springer Berlin Heidelberg, 2012.

[6] S. Che, M. Boyer, et al. A Performance Study of General-Purpose Applications on Graphics Processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10): 1370 – 1380, 2008.

[7] Chrysos, G. Intel Xeon Phi Coprocessor - the Architecture. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, 2012.

[8] NVIDIA Corporation. CUDA C Best Practices Guide Version 7.0, 2015.

[9] NVIDIA Corporation. CUDA C Programming Guide Version 7.0, 2015.

[10] L. Dawson and I. Stewart. Improving Ant Colony Optimization performance on the GPU using CUDA. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1901–1908, 2013.

[11] T. Deakin, J. Price, et al. GPU-STREAM (UoB HPC Group). https://github.com/UoB-HPC/GPU-STREAM, 2015.

[12] I. Demeshko, H.C. Edwards, M.A. Heroux, et al. Kokkos implementation of Albany: a performance-portable finite element application. Technical Report SAND2014-18789A, Sandia National Laboratories, 2014.

[13] R. Dietrich, F. Schmitt, A. Grund, and D. Schmidl. Performance Measurement for the OpenMP 4.0 Offloading Model. In *Euro-Par 2014: Parallel Processing Workshops*, pages 291–301. Springer, 2014.

[14] J. Dongarra et al. The International Exascale Software Project roadmap. *International Journal of High Performance Computing Applications*, 2011.

[15] H C. Edwards, C.R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[16] H.C. Edwards, C. Trott, and D. Sunderland. Kokkos Tutorial: A Trilinos Package for Manycore Performance Portability. Available from: http://trilinos.org/oldsite/events/trilinos_user_group_2013/presentations/2013-11-TUG-Kokkos-Tutorial.pdf, 2013.

[17] J. Godwin, J. Holewinski, et al. High-performance Sparse Matrix-vector Multiplication on GPUs for Structured Grid Computations. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 47–56, New York, NY, USA., 2012. ACM.

[18] Z. Grewe, D. Wang and M.FP. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

[19] Khronos OpenCL Working Group. The OpenCL Specification Version 1.2, 2015.

[20] R. Hazra. Accelerating Insights in the Technical Computing Transformation. In *International Supercomputing Conference*, 2014.

[21] M.A. Heroux, D.W. Doerfler, et al. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[22] R.D. Hornung and J.A. Keasler. A Case for Improved C++ Compiler Support to Enable Performance Portability in Large Physics Simulation Codes. Technical Report LLNL-TR-635681, Lawrence Livermore National Laboratory, 2013.

[23] R.D. Hornung, J.A. Keasler, et al. The RAJA Portability Layer: Overview and Status. Technical Report LLNL-TR-661403, Lawrence Livermore National Laboratory, 2014.

[24] Kanter, D. Microservers must Specialize to Survive. http://www.realworldtech.com/microservers/, 2013.

[25] C. Kessler, U. Dastgeer, S. Thibault, R. Namyst, et al. Programmability and Performance Portability Aspects of Heterogeneous Multi-/Manycore Systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 1403–1408. IEEE, 2012.

[26] P. Kogge and J. Shalf. Exascale Computing Trends: Adjusting to the "New Normal" for Computer Architecture. *Computing in Science & Engineering*, 15(6):16–26, 2013.

[27] A. Kukanov, V. Polin, and M.J. Voss. Flow Graphs, Speculative Locks, and Task Arenas in Intel® Threading Building Blocks. *Graduate from MIT to GCC Mainline*, page 15, 2014.

[28] D. Lee, I. Dinov, B. Dong, et al. CUDA Optimization Strategies for Compute- and Memory-Bound Neuroimaging Algorithms. *Computer Methods and Programs in Biomedicine*, 106(3):175 – 187, 2012.

[29] V.W. Lee, C. Kim, J. Chhugani, et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, 2010.

[30] C. Liao, Y. Yan, B.R. de Supinski, D.J. Quinlan, and B. Chapman. Early Experiences with the Owill primarily be targettable aopenMP Accelerator Model. In *OopenMP in the Era of Low Power Devices and Accelerators*, pages 84–98. Springer, 2013.

[31] A.C. Mallinson, D.A. Beckingsale, W.P. Gaudin, J.A. Herdman, and S.A. Jarvis. Towards Portable Performance for Explicit Hydrodynamics Codes. 2013.

[32] J.D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA)*, pages 19–25, 1995.

[33] S. McIntosh-Smith and D. Curran. Evaluation of a Performance Portable Lattice Boltzmann Code Using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, pages 2:1–2:12, New York, NY, USA., 2014. ACM.

[34] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price. On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures. In *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 53–75. Springer International Publishing, 2014.

[35] McIntosh-Smith, S. and Price, J. and Sessions, R.B. and Ibarra, A.A. High performance in silico virtual drug screening on many-core processors. *International Journal of High Performance Computing Applications*, 2014.

[36] A. Munshi, B. Gaster, T.G. Mattson, and D. Ginsburg. *OpenCL Programming Guide*. Pearson Education, 2011.

[37] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210, 2014.

[38] E.d Labarta J. Ozen, G. Ayguadé. On the Roles of the Programmer, the Compiler and the Runtime System When Programming Accelerators in OpenMP. In *Using and Improving OpenMP for Devices, Tasks, and More*, pages 215–229. Springer, 2014.

[39] D. Peng, W. Rick, L. Piotr, T. Stanimire, P. Gregory, and D. Jack. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391 – 407, 2012.

[40] K. Raman. Optimizing Memory Bandwidth on Stream Triad. https://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad, 2013.

[41] A. Rane and D. Stanzione. Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems. In *Proceedings of 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[42] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - a High Level Linear Algebra Library for GPUs and Multi-Core CPUs. *Proc. GPUScA*, pages 51–56, 2010.

[43] K. Rupp, P. Tillet, F. Rudolf, et al. Performance Portability Study of Linear Algebra Kernels in OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, pages 8:1–8:11, New York, NY, USA, 2014. ACM.

[44] O.P. Sachan. Intel Compilers for Linux*: Compatability with GNU Compilers. Technical report, 2008.

[45] E. Saule, K. Kaya, and Ü.V. Çatalyürek. Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi. In *Parallel Processing and Applied Mathematics*, pages 559–570. Springer, 2014.

[46] A. Sidelnik, S. Maleki, B. L Chamberlain, et al. Performance portability with the Chapel language. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 582–594. IEEE, 2012.

[47] J.E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[48] J.A. Stratton, H. Kim, T.B. Jablin, and W.W. Hwu. Performance Portability in Accelerated Parallel Kernels. *Center for Reliable and High-Performance Computing*, 2013.

[49] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz. Comparative Performance Analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: A Case Study from Microscopy Image Analysis. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1063–1072. IEEE, 2014.

[50] P. Tillet, K. Rupp, S. Selberherr, and C. Lin. Towards Performance-Portable, Scalable, and Convenient Linear Algebra. *Talk: HotPar*, 2013.

[51] C.R. Trott, M. Hoemmen, S.D. Hammond, and H.C. Edwards. Kokkos Programming Guide. Technical Report SAND2015-4178, Sandia National Laboratories, 2015.

[52] UKMAC. UK Mini-App Consortium: TeaLeaf. http://uk-mac.github.io/TeaLeaf, 2015.

[53] University of Illinois at Urbana-Champaign. The IMPACT Research Group: Parboil Benchmarks. http://impact.crhc.illinois.edu/Parboil/parboil.aspx, 2015.

[54] S. Wienke, C. Terboven, James C. Beyer, and M.S. Müller. A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing. In *Euro-Par 2014 Parallel Processing*, pages 812–823. Springer, 2014.

[55] X. Yan, X. Shi, L. Wang, and H. Yang. An OpenCL Micro-Benchmark Suite for GPUs and CPUs. *The Journal of Supercomputing*, 69(2):693–713, 2014.

[56] Y. Zhang, M. Sinclair II, and A.A. Chien. Improving performance portability in OpenCL programs. In *Supercomputing*, pages 136–150. Springer, 2013.